CONCEVOIR EN ORIENTE OBJET AVEC ATGL/OOSD/TOOD.c

TABLE DES MATIERES

1. PRESENTATION GENERALE DU DOCUMENT 2
1.1. Rôle du document 2
1.2. Aperçu sur le document 2

2. DOCUMENTS CITES EN REFERENCE 4

3. BASES DE LA CONCEPTION ORIENTEE OBJETS 5
3.1. Elément (terme générique) / Element (generic word) 5
3.2. Vocabulaire de base "objet" / "object" basic vocabulary 5
3.2.1. Objet / Object 5
3.2.2. Classe d'objets / Object classe 7
3.3. Qualificatifs / qualificative expressions 7
3.3.1. Passif–Actif / Passive–Active 7
3.3.2. Partagé / shared 8
3.3.3. Objet composite–Objet de base / Composite object–basic object 8
3.3.4. Strict – non strict / strict – non strict 8
3.4. Relations / Relationships 9
3.5. Les entités / Entities 9
3.5.1. Entité (terme générique) / Entity (generic word) 9
3.5.2. Introduction aux entités / Entities overview 10

4. LES GENRES D'ENTITES,
leur représentation, leur traduction en code,
leur signification dans le contexte DOD 13
4.1. Généralités sur les entités 13
4.1.1. Organisation des entités 13
4.1.2. Représentation graphique générale 14
4.1.3. Généralités sur le lien entre les entités et les CSCs/CSUs 15
4.1.4. Généralités sur l'outil 16
4.1.5. Généralités sur la traduction en C ANSI 16
4.1.5.1. Le choix Ansi 16
4.1.5.2. Les innovations du langage C ansi 16
4.1.5.3. Passage à la normalisation 18
4.1.5.4. Avenir 18
4.1.5.5. Les entités de la conception 18
4.2. La machine 19
4.2.1. Introduction 19
4.2.2. Interface 20
4.2.3. Environnement 22
4.2.4. Corps 23
4.2.5. Machines dégénérées 24
4.2.6. Traduction en C ANSI 24

4.3. Processus 25
4.3.1. Introduction 25
4.3.2. Interface 25
4.3.3. Environnement 28
4.3.4. Corps 28
4.3.5. Traduction en C ANSI 28
4.4. Sous–application 29
4.4.1. Introduction 29
4.4.2. Interface 30
4.4.3. Environnement 32
4.4.4. Corps 32
4.4.5. Sous–application dégénérée 34
4.4.6. Traduction en C ANSI 35
4.4.6.1. Sous–application (stricte) 35
4.4.6.2. Sous–application non stricte 36
4.5. Le type abstrait de données 37
4.5.1. Introduction 37
4.5.2. Interface 38
4.5.3. Environnement 39
4.5.4. Corps 40
4.5.5. Types abstraits dégénérés 41
4.5.6. Traduction en C ANSI 41
4.5.6.1. Fichier spécification "MotLong.h" 41
4.5.6.2. Fichier corps "MotLong.c" 42
4.6. Type abstrait de processus 45
4.6.1. Introduction 45
4.6.2. Interface 46
4.6.3. Environnement 46
4.6.4. Corps 46
4.7. Collection 47
4.7.1. Introduction 47
4.7.2. Interface 47
4.7.3. Environnement 48
4.7.4. Corps 48
4.7.5. Traduction C ANSI 50

5. LIENS AVEC LE SDD 51
5.1. Phasage : conception préliminaire/conception détaillée 51
5.2. Schémas de OOSD dans le SDD 51
5.3. Génération du SDD 52

6. DEMARCHE 54
6.1. Introduction 54
6.2. Quelques remarques introductives 55
6.2.1. ... A propos du séquencement des activités : 55
6.2.2. ... A propos de la transition spécification/conception, ... ou désorganiser pour réorganiser : 55
6.2.3. ... A propos de la justification des choix : 55
6.2.4. Quand peut–on réellement commencer l'organisation du logiciel ? 56
6.3. La démarche de conception 57
6.3.1. Niveau système 57
6.3.2. Niveau CSCI 58
6.4. Fixer l'interface de l'application 63
6.4.1. Objectif : 63
6.4.2. Guide : 63
6.5. Identifier les candidats objets 65
6.5.1. Identification proprement dite des objets 65
6.5.1.1. Objectif : 65
6.5.1.2. Guide : 65
6.5.1.2.1. Applications de taille importante 65
6.5.1.2.2. A partir de SA–RT : 66
6.5.1.2.3. Cas des spécifications textuelles 70
6.5.2. Identification des fonctionnalités 71
6.5.2.1. Objectif : 71
6.5.2.2. Guide : 71
6.5.2.2.1. Introduction 71
6.5.2.2.2. A partir de SA–RT 71
6.5.2.2.3. A quel objet associer l'action ? 72
6.5.2.2.4. A partir d'une spécification textuelle 73
6.5.3. Complément d'analyse sur les objets 73
6.5.3.1. Objectif : 73
6.5.3.2. Guide : 73
6.5.4. Complément d'analyse sur les fonctionnalités 74
6.5.4.1. Objectif : 74
6.5.4.2. Guide : 75
6.6. Répartir les candidats–objets en entités "nuages" 76
6.6.1. Objectif : 76
6.6.2. Guide : 76
6.7. Fixer l'interface des sous–applications 78
6.7.1. Objectif : 78
6.7.2. Guide : 78
6.8. Analyser l'ordonnancement (et donc fixer le parallélisme) 80
6.8.1. Objectif : 80
6.8.2. Guide : 80
6.8.3. A partir de SA–RT : 80
6.8.4. Genres classiques de processus 81
6.9. Choix de la description des informations (machine ou type abstrait?) 82
6.9.1. Objectif : 82
6.9.2. Guide : 82

7. PRINCIPES DE CONCEPTION 84
7.1. Décomposition / Decomposition 84
7.2. Modularité / Modularity 84
7.3. Abstraction / Abstraction 85
7.4. Masquage / Information Hiding 86
7.5. Hiérarchie /Hierarchy 87
7.6. Typage 89
7.7. Parallélisme 90

8. GLOSSAIRE 92

LISTE DES FIGURES

Figure 3.1. 10
Figure 4.1. interface d'entité 14
Figure 4.2. Corps d'entité composite 15
Figure 4.3. Interface d'entité réutilisée 15
Figure 4.4. modèle d'interface de machine 20
Figure 4.5. Exemple d'interface de machine 21
Figure 4.6. Environnement de la machine DICTIONNAIRE 22
Figure 4.7. Exemple de représentation graphique d'un corps de machine 23
Figure 4.8. Modèle de l'interface d'un processus 26
Figure 4.9. Exemple de l'interface d'un processus 26
Figure 4.10. Opération à haut degré de synchronisation 27
Figure 4.11. Opération à bas degré de synchronisation 27
Figure 4.12. Opération asynchrone 27
Figure 4.13. Interface d'une sous–application 31
Figure 4.14. Environnement d'une sous–application 32
Figure 4.15. Exemple de corps de sous–application se décomposant en sous–applications 33
Figure 4.16. Exemple de corps de sous–application se décomposant en machines 33
Figure 4.17. Exemple de corps de sous–application non stricte. 34
Figure 4.18. exemple d'interface de type abstrait de données 38
Figure 4.19. Forme de l'interface d'un type abstrait de processus 46
Figure 4.20. Exemple d'interface d'une collection se décomposant en collections 48
Figure 4.21. Exemple d'interface d'une collection se décomposant en type abstraits 48
Figure 4.22. Corps de collection composé de collections 49
Figure 4.23. Corps d'une collection composé de types abstraits 49
Figure 6.1. Schéma général de gestion de la réutilisation au niveau système 57
Figure 6.2. Vision synthétique de la démarche en termes de niveaux de détail 58
Figure 6.4. 67
Figure 6.5. 68
Figure 6.6. Diagramme de flot de données SA–RT 72
Figure 7.2. 88
Figure 8.1. 95
Figure 8.2. 96

AVERTISSEMENT – ERRATUM

L'utilisation des commentaires dans les graphiques pour indiquer la nature des sous–applications (SA : sous–application active, SAP : sous–application passive, NS : sous–application non stricte active et NSP : sous–application non stricte passive) est à éviter ou à utiliser avec la plus grande précaution.

Cette indication concerne les pages 31, 33 (dans les figures) et 34 (dans la figure).

En effet, la personnalisation de l'outil OOSD/ADE à TOOD.c ne s'appuiera pas sur ce choix (pour des raisons de difficultés de manipulations par l'utilisateur et lors des contrôles effectués par l'outil), et proposera une autre différenciation entre les machines et les sous–applications.

1. PRESENTATION GENERALE DU DOCUMENT

1.1. Rôle du document

Ce document a pour objectif de définir l'approche THOMSON–CSF dans la conception de logiciels temps réel avec l'approche orientée par les objets, par l'utilisation de l'outil ADE et dans le contexte normatif de la DOD–STD–2167A.

Issu et destiné aux concepteurs de logiciels du groupe THOMSON–CSF dans la conception de logiciels, *ce document est et doit être le résultat de la factorisation et de la capitalisation des compétences du groupe : il doit donc être enrichi, chaque fois qu'un concepteur du groupe peut proposer une technique, un conseil, une remarque complétant le document et apportant une aide aux autres concepteurs.*

Ce document doit donc vivre, d'autant que la présente première version ne prétend pas couvrir l'ensemble des besoins des concepteurs, mais est seulement la première étape vers cette capitalisation de la compétence.

Ce document est une version intermédiaire entre un manuel de référence et un manuel d'utilisation; il contient donc des explications, mais celles–ci ne prétendent pas être complètes : l'objectif est de permettre à des concepteurs qui doivent commencer une conception immédiatement d'avoir la base de définitions et d'explications nécessaires.

Ce document décrit le langage, la démarche et l'outil et leur utilisation pour les projets utilisant la conception orientée par les objets.

1.2. Aperçu sur le document

Le chapitre 3 est une introduction à la conception orientée par les objets et au vocabulaire utilisé.

Le chapitre 4 décrit les différentes entités qui sont les implémentations des objets/modes d'objets et sont les éléments de base de la méthode de conception : sous–application, machine, type abstrait, processus, collection et leurs différentes variantes, en particulier génériques.

Le chapitre 5 effectue le lien avec la norme DOD–STD–2167A : correspondance entre les entités de conception et les composants de la DOD, modèle de constitution du document de conception défini par cette norme (SDD).

Le chapitre 6 détaille les différentes étapes de la démarche de conception orientée par les objets.

Dans le chapitre 7 sont introduits les principes de base de conception : abstraction, modularité ... ainsi que leurs justifications. Bien que ce chapitre ne soit pas propre à la conception orientée par les objets, sa lecture est conseillée pour le concepteur avant la mise en application de la méthode.

Le chapitre 8 est un glossaire des termes utilisés.

Enfin, tout ce qui concerne l'outil est marqué par une barre verticale noire en marge gauche.

2. DOCUMENTS CITES EN REFERENCE

[Z61–102] Vocabulaire de la qualité du logiciel (juillet 1985)

[NF L 00–007] Industrie aéronautique et spatiale – Vocabulaire – Termes généraux (mars 1987)

[AQUAP 15] Glossaire de termes utilisés dans les STANAG AQ et les AQAP (mars 1986)

[ANSI/IEEE, Std 729–1983] IEEE Standard Glossary of Software Engineering Terminology, 1983

[DOD–STD–2167A] Military Standard – Defense system software development, 1988

[VQ 02–1990 E/F] Vocabulaire des termes normalisés usuels de la qualité – 1990 – THOMSON–CSF

[BOOCH–91] Object Oriented Design with applications, G. BOOCH, 1991

[ZILLES–84] Types, Algebras and Modelling, in On Conceptual Modeling Perspectives from Artificial Intelligence, Databases, and Programming Languages, 1984

3. BASES DE LA CONCEPTION ORIENTEE OBJETS

3.1. Elément (terme générique) / *Element (generic word)*

Terme générique correspondant à tout élément ou partie d'élément (du monde réel, du monde informatique), clairement identifiable, c'est–à–dire :

- possédant un nom propre,
- dont on peut caractériser les limites/frontières, c'est–à–dire ce qui en fait partie par rapport à ce qui n'en fait pas partie,
- qui est manipulable de manière indépendante.

Par exemple, dans le monde réel, une règle centimétrique est un élément. Par contre, le morceau de règle allant de la troisième marque centimétrique jusqu'à la dix–septième n'est pas un élément car il n'a pas de nom propre et surtout il n'est pas manipulable de manière indépendante.

Dans le monde informatique, un paquetage Ada, un fichier C, une procédure ou une fonction, une donnée ... sont des éléments. Par contre, la partie de code allant de la 3 ème ligne à la dix–septième ligne n'est pas un élément (sauf bien entendu, si cela correspond exactement à la zone de définition d'une procédure, mais ce n'est alors pas la bonne manière de l'identifier).

3.2. Vocabulaire de base "objet" / *"object" basic vocabulary*

Il existe trois sortes d'éléments à la base d'une méthode de conception orientée par les objets : l'objet, la classe d'objets et les génériques.

3.2.1. Objet / *Object*

A quelques nuances près, nous nous alignons sur les concepts objet de base sur ce que propose G.BOOCH, car ces définitions ont l'avantage d'être connues et répandues.

[BOOCH–91] :

Un objet a

- un *état*,
- un *comportement*,
- une *identité*.

La structure et le comportement d'objets semblables sont définis dans leur classe

commune[1].

Un objet est une abstraction du monde réel : cette abstraction peut être "calquée" sur le concret (objet table) ou correspondre à une modélisation du concret (dimension de la table) : le concept de taille (ou distance) "n'existe pas" dans le monde concret au sens où elle n'est pas palpable, mais correspond à une modélisation – mathématique, physique ...– du monde concret sur lequel on peut raisonner, calculer... C'est donc une abstraction du monde réel qui sera représentée dans le monde informatique comme un objet.

Si un objet est unique, il lui sera associé une description sous forme d'entité (cf 3.5.), sinon sa description sera déduite de son modèle, et donc de l'entité de description de ce modèle.

Etat d'un objet / *Object state*

[BOOCH–91] :

> L'*état* d'un objet englobe toutes les propriétés (en général statiques) d'un objet, plus les valeurs courantes (en général dynamiques) de chacune de ces propriétés.

Le terme propriété correspond à une caractéristique, ou à une information "contenue" dans l'objet.

[BOOCH–91] :

> La *structure* d'un objet est la représentation concrète de son état.

Une autre manière de présenter ce qui touche à l'état d'un objet est la suivante :

> un objet "contient" une certaine quantité d'informations. Plus précisement, il est composé de "réceptacles" d'informations (différence entre le contenant et le contenu, ou entre une variable et sa valeur), à partir desquels il est possible d'extraire des informations (soit les informations sont contenues dans les réceptacles, soit elles sont déduites des informations contenues).
>
> L'état d'un objet correspond à la valeur de l'ensemble de ces informations. La structure de l'objet est la description de l'ensemble des receptacles.
>
> L'état d'un objet représente à la fois les caractéristiques statiques (du genre structure du contenant ou propriétés intrinsèques de l'objet : un poisson vit dans l'eau) et les valeurs contenues dans le contenant ou déductibles du contenant.

Comportement d'un objet / *Object behaviour*

[BOOCH–91] :

> Le *comportement* définit comment un objet agit ou réagit, en termes de changements d'états et d'échanges de messages.

1. On ne reprend pas la définition complète de G.BOOCH qui ajoute : les termes instance et objet sont synonymes.
En effet, il part du principe que tout objet appartient à une classe, ce qui permet d'unifier les définitions. Dans cette approche, si un objet est unique dans une application, il est assez fastidieux de décrire la classe, puis d'indiquer que cet objet est l'instance de la classe. En fait, ce qui est proposé est de décrire directement l'objet, mais "conceptuellement", cela correspond à décrire une classe anonyme dont l'objet est l'unique instance.
Ce genre d'argumentation est intéressante d'un point de vue théorique, mais n'apporte rien au savoir–concevoir, et n'a donc pas été introduite ici. De plus, cela peut amener des confusions sur le partage de telles informations, dans un contexte parallèle.

Un échange de message dans ce contexte correspond à un appel d'opération, avec ou sans "paramètres". Une *opération* est une action qu'un objet "lance" sur un autre objet en vue de provoquer une réaction. Echange de message et appel d'opération sont synonymes.

La description de l'ensemble des opérations qui peuvent être exécutées sur l'objet définit le comportement possible de l'objet (vision statique du comportement).

Identité d'un objet / *Object identity*

L'identité sera faiblement utilisée dans cette approche. G. BOOCH la définit ainsi :

[BOOCH–91] :

> L'*identité* est la propriété d'un objet qui le distingue de tous les autres objets.

L'identité correspond au contenant et au contenu d'un objet, c'est–à–dire des informations appartenant à l'objet. Attention, le nom d'une variable n'est pas l'identité de l'objet, mais un moyen de désigner pendant un temps donné l'objet par ce nom. En simplifiant, le nom d'une variable est un "pointeur sur un objet" alors que l'adresse de l'objet est son identité.

3.2.2. Classe d'objets / *Object classe*

[BOOCH–91]

> Une *classe* est un ensemble d'objets qui ont une structure et un comportement communs.

Une classe est le modèle d'un ensemble d'objets et contient donc le descriptif de cette structure et de ce comportement communs à tous les objets de l'ensemble.

Un objet qui est conforme au modèle qu'est la classe est une *instance* de la classe.

Partager une classe ne pose aucun problème dans une application parallèle (cf 7.7. pour la définition du parallélisme) puisque la classe ne contient pas de données en elle–même (ce sont les instances qui en contiennent).

A toute classe sera associée une description sous forme d'entité (cf 3.5.).

3.3. Qualificatifs / *qualificative expressions*

Avertissement : les qualificatifs suivants s'appliquent aussi bien aux objets, classes d'objets et génériques qu'aux entités qui les décrivent. La définition qui en est donnée ici est appliquée aux objets, elle sera étendue dans le paragraphe sur les entités pour les appliquer aux entités.

3.3.1. Passif–Actif / *Passive–Active*

Un objet est *passif* si son activité est uniquement composée de réactions provoquées par les appels d'opérations effectués par son environnement.

Plus simplement : il ne fait rien si on ne lui demande rien !

Un objet est *actif* s'il a une activité simultanée (ou parallèle) avec le reste de l'application, en dehors des réactions provoqués par les appels d'opérations effectués par son environnement.

Un objet qui n'est pas passif est actif et réciproquement. Un objet actif est dit avoir un comportement propre. Il contient (au moins) un *thread of control* qui lui est propre.

3.3.2. Partagé / *shared*

Un objet est *partagé* si ses opérations peuvent être appelées par plusieurs objets actifs[2].

3.3.3. Objet composite–Objet de base / *Composite object–basic object*

Un objet est *composite* s'il se décompose en d'autres objets (donc de plus bas niveau de détail) et si sa description se décompose en la description des objets contenus (donc l'entité qui le décrit se décompose en entités qui décrivent chacun l'un des composants de l'objet).

Un objet *de base* est un objet non composite, c'est–à–dire que l'entité qui le décrit ne se décompose pas en (sous–)entités. Par contre, l'entité qui le décrit peut bien entendu utiliser d'autres entités descriptifs d'objets ou de modèles d'objets.

3.3.4. Strict – non strict / *strict – non strict*

Un objet composite est strict s'il n'est que l'assemblage des objets qui le compose; autrement dit, il n'introduit rien d'autre que ses objets composants. En particulier, il ne contient donc ni structure ni comportement qui ne soit attaché à l'un de ses objets composants.

L'interface d'un objet composite strict est la partie des interfaces des objets composants que l'objet composite rend visible. Une opération située sur l'interface d'un objet composite strict n'est donc qu'une opération d'un des objets composants qui est rendue visible par l'objet composite.

Le corps de l'entité qui le décrit ne contient donc que la description de ses objets composants et de leurs interfaces, ainsi que la correspondance entre chaque élément (opération, ...) de l'interface de l'objet composite et un élément d'un objet composant (l'élément de l'objet composant est un élément rendu visible par l'objet composite).

Un objet composite non strict est un objet composite dont une partie de la structure et/ou du comportement n'est pas définie par l'un des objets composants mais est lié à l'objet composite. En particulier, une ou plusieurs informations peuvent être associées à l'objet composite sans appartenir à aucun des objets composants. Une ou plusieurs opérations peuvent avoir une définition associée à l'objet composite (et s'appuyer ou non sur les opérations des objets composants).

2. Plus précisément, un objet est partagé si ses opérations peuvent être appelées par plusieurs objets actifs de base, ou encore s'il peut être "traversé" par plusieurs *thread of control*.

Le corps de l'entité qui le décrit contient donc la description de ses objets composants et de leurs interfaces, ainsi que des éléments de structure et des opérations qui n'appartiennent à aucun composant.

3.4. Relations / *Relationships*

Avertissement : de même que pour les qualificatifs, les relations introduites ci–après s'appliquent aussi bien aux objets, classes d'objets et génériques qu'aux entités qui les décrivent.

Dans la première version de cette approche dont les langages cibles ne sont pas les langages à objets (PPO), les relations sont des trois genres suivants :

– relation "utilisation" :
  * un objet utilise un autre objet s'il appelle au moins une opération de celui–ci;
  * un objet utilise un modèle (classe ou générique) d'objets s'il utilise (de manière statique ou dynamique – i.e. localement à une opération de l'objet–) au moins un objet instance du modèle;
  * un modèle d'objets A utilise un modèle d'objets B si tout objet instance de A utilise (ou plus exactement : peut utiliser) le modèle B;

– relation "est partie de" (décomposition) :
  * un objet est partie d'un autre objet ... s'il est contenu dans celui–ci; son état fait partie de l'état de l'objet englobant;

– relation "est un" (ou "est instance de") :
  * un objet instance d'une classe signifie que la partie statique de son état et son comportement sont décrits dans le modèle qu'est la classe;
  * un objet instance d'une objet générique signifie que la partie statique de son état et son comportement sont partiellement décrits dans le modèle qu'est l'objet générique et que le complément de description a lieu lors de l'instanciation;
  * une classe instance d'une classe générique signifie que la partie statique de l'état et le comportement de ses instances sont partiellement décrits dans le modèle qu'est la classe générique et que le complément de description a lieu lors de l'instanciation.

3.5. Les entités / *Entities*

3.5.1. Entité (terme générique) / *Entity (generic word)*

Dans l'approche COO, une entité est un élément qui est le support de description d'un objet si celui–ci a une description propre, ou à la description d'un modèle d'objets (classe ou générique) ou d'une collection de tels modèles. Elle correspond donc à une unité de programmation, ou à un ensemble de telles unités.

Une entité est décrite en deux parties, son *interface et son corps*. (cf masquage)

3.5.2. Introduction aux entités / *Entities overview*

La conception étant la description de l'architecture du logiciel, elle s'exprime en termes d'entités, ces entités ayant une unité conceptuelle construite autour de l'objet (ou d'un modèle d'objets ou d'une collection comme c'est indiqué ci–après).

La séparation entre les "idées de base", qui s'appuient sur ce qui existe dans la littérature reconnue sur le sujet, et la "mise en oeuvre industrielle" se situe au niveau des entités qui vont représenter l'implémentation de l'approche de conception orientée par les objets.

L'identification des entités est basée sur une analyse du risque en particulier lié au parallélisme propre à tout système temps réel.

Le risque lié aux différents genres d'objets n'étant pas de même nature, l'organisation de l'architecture du logiciel sépare tant que faire se peut les genres d'objets de criticité différentes.

Chaque genre d'entité correspondant à l'implémentation d'un objet/modèle/collection de modèles, le nom donné à l'entité permet également de retrouver les caractéristiques de l'objet/modèle/collection de modèles. Le nom donné à une entité est souvent plus simple à manipuler que le nom de l'objet (que l'entité implémente) muni de toutes ses caractéristiques; ainsi, il est plus simple de parler de "processus" que d'"objet actif de base", ou de parler de "sous–application" plutôt que d'"objet composite strict" ...

Le reste du document s'appuiera largement sur ces termes, car c'est autour des entités que la conception s'architecture.

La figure 3.1. suivante présente les noms donnés aux entités décrivant les objets/modèles/collections de modèles, selon les concepts qui ont été introduits.

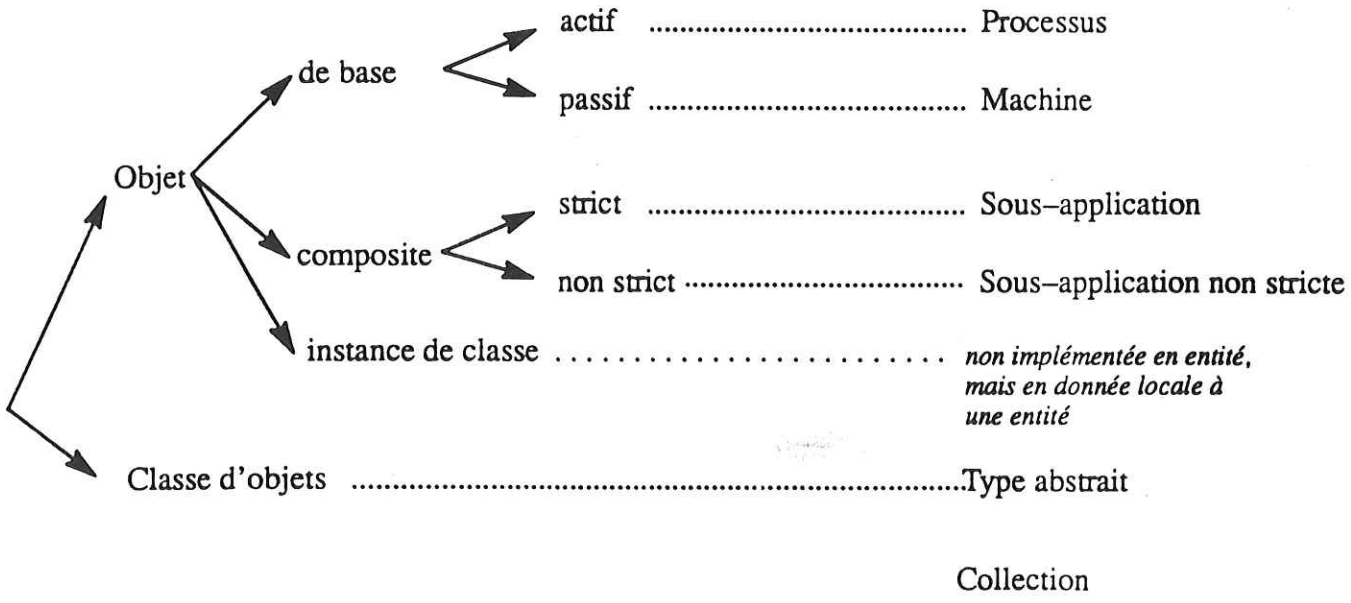


Figure 3.1.

Une entité particulière, non "accrochée" à un objet/modèle apparaît : la collection. Cette entité est nécessaire pour l'organisation des modèles (types abstraits et génériques). Une collection peut se décomposer en (sous–)collections... La collection correspond à la notion de bibliothèque.

Le *processus* identifie un (et un seul) *thread of control*, c'est à dire une séquence d'activités qui s'exécutent simultanément (virtuellement ou réellement) avec les autres séquences d'activités.

La *machine* est l'implémentation d'un objet qui est passif, c'est–à–dire qui ne sait que répondre à des stimulis.

Une *sous–application* est un assemblage strict d'entités (pas de contenu autre que les entités). Le corps de la sous–application ne contient donc que la description de ses entités composantes et de leurs interfaces, ainsi que l'attachement de chaque opération de la sous–application à une opération de l'objet composant qui prend en charge l'opération.

Un *sous–application non stricte* est un assemblage non strict d'entités (une partie de son contenu ne se retrouve dans aucune des entités qui le composent). Le corps de l'sous–application non strict contient donc la description de ses entités composantes et de leurs interfaces, ainsi que les éléments de structure qui n'appartiennent à aucun composant, et les opérations qui correspondent en général à une logique d'appel des opérations des objets composants.

Les caractéristiques liées aux objets/modèles/collections se transposent sur les entités. Ainsi :

- une entité est *de base* si elle ne se décompose pas en (sous–)entités; au contraire, une entité est *composite* si elle elle se décompose en (sous–)entités.
- une entité est *active* si elle implémente un objet actif; elle est *passive* si elle implémente un objet actif.
- une entité est *partagée* si l'objet qu'elle implémente est partagé – ou, ce qui est équivalent – si le jeu d'opérations qu'elle offre dans son interface peut être appelé par plusieurs objets actifs (de base); ce terme sera en général dédié à la qualification des machines.

Les termes strict et non strict s'associent de manière bijective aux entités sous–application et sous–application non strict, respectivement.

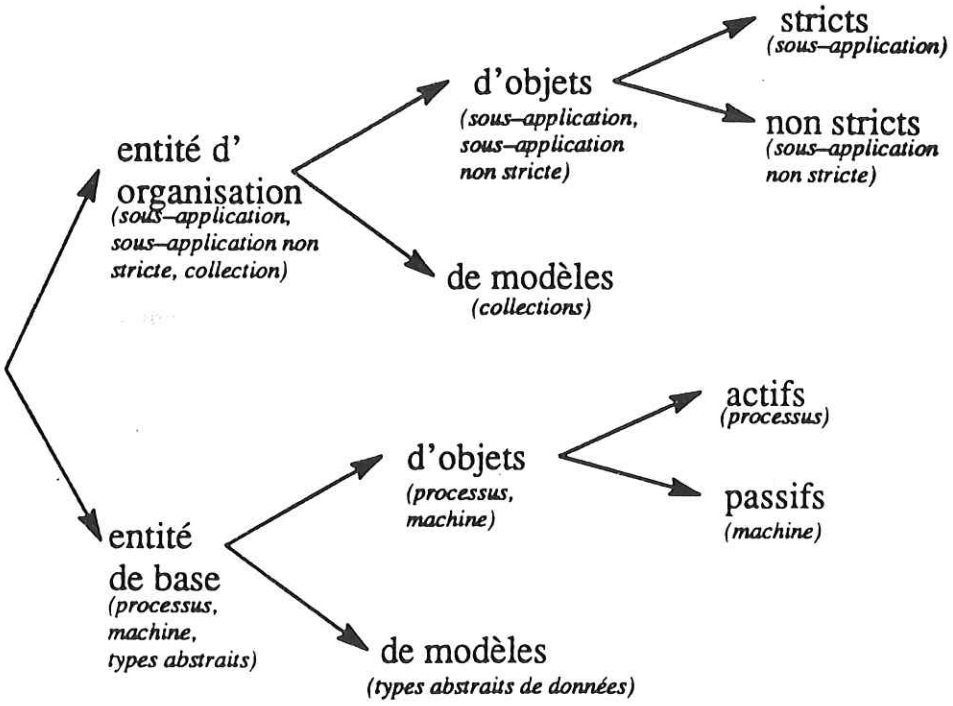
Les entités "héritent" des caractéristiques liées aux objets/modèles/collections qu'elles implémentent. Ainsi :

- un processus est une entité de base active (par définition),
- une machine est une entité de base passive (par définition); elle peut être ou non partagée;
- une sous–application est une entité composite stricte (par définition); elle peut être active ou passive suivant qu'elle contient ou non au moins un processus, respectivement;

- un sous-application non strict est une entité composite non stricte (par définition); elle peut être active ou passive suivant qu'elle contient ou non au moins un processus, respectivement;
- une collection est une entité composite. Les autres caractéristiques n'ont pas de sens pour la collection.

Une classe d'objets composites est implémentée comme une entité type abstrait (de base) utilisant les entités implémentant les objets composants de la classe composite (éventuellement regroupés dans une collection). Un type abstrait est donc toujours considéré comme étant de base.

Une autre classification des entités basée sur la criticité ou sur la documentation associée est la suivante :



La séparation des entités d'organisation des entités de base vient de leur différence de complexité et de documentation associée.

La séparation entre les entités de description d'objets et les entités de description de modèles vient de la différence de criticité dans le cas du partage : le partage d'un objet est critique (conflit d'accès) alors que le partage de modèles ne l'est pas.

La séparation entre entités de description d'objets passifs et actifs est bien entendu liée à leur différence fondamentale pour le parallélisme.

4. LES GENRES D'ENTITES, leur représentation, leur traduction en code, leur signification dans le contexte DOD

4.1. Généralités sur les entités

4.1.1. Organisation des entités

Dans l'approche orientée par les objets, l'entité est la description d'un objet ou d'un modèle d'objets, ou encore d'une collection de modèles. Pour simplifier l'expression des définitions ou des explications, dans ce paragraphe, on considèrera que l'entité est la description "d'un objet", ce qui évitera d'écrire ".. de l'objet ou du modèle d'objets ou de la collection de modèles d'objets ...", ou lorsque l'on parle par exemple d'une opération décrite dans l'entité qu'elle représente une action "applicable à l'objet ou à toute instance du modèle d'objets suivant le cas".

L'entité est défini en deux étapes:

- la première consiste à établir l'*interface* de l'entité, c'est à dire tout ce qui est utile pour pouvoir "utiliser" l'entité : c'est la vision extérieure de l'entité ou vision "boîte noire".
- la seconde étape consiste à décrire le *corps* de l'entité : c'est l'ensemble des informations nécessaires à la description de l'objet mais qu'il est inutile de connaître pour une bonne utilisation de l'entité qui le décrit. Alors que l'interface définit les actions sur l'objet, le corps décrit comment ces actions sont réalisées ainsi que la structure interne de l'objet éventuellement en terme de ressources. C'est la vision "boîte blanche".

Un diagramme est possible pour chacune de ces deux visions. Il ne sera pas nécessaire de décrire le corps des entités de base par un diagramme.

A ces diagrammes interface et corps, est ajouté un diagramme *environnement.*

- l'environnement de l'entité indique les entités extérieures qui sont nécessaires pour construire entièrement l'entité. Pour construire l'interface du entité, seuls des types abstraits (ou des collections de tels types abstraits) peuvent être nécessaires. Pour construire le corps, des entités de tout genre peuvent être nécessaires.

Pour un objet *nom_d_objet* on peut avoir trois diagrammes :

- un diagramme d'interface qui a le même nom que l'objet;
- un diagramme d'environnement qui s'appelle *nom_d_objet*_env;
- un diagramme de corps qui s'appelle *nom_d_objet*_body.

Ils sont atteints respectivement par les boutons push, push_env et push_body.

4.1.2. Représentation graphique générale

L'interface d'une entité peut contenir:

* des opérations que l'entité met à la disposition de l'environnement pour que celui–ci puisse agir sur l'entité;
* des types de données;
* des constantes.

Attention : la présence de ces éléments d'interface dépendent du genre d'entité. Selon les genres d'entités, ces éléments peuvent/doivent ou non apparaître dans l'interface.

En aucun cas les descriptions des objets composants (qu'ils soient instances de classes d'objets – donc décrits sous forme de données – ou objets uniques – donc décrits sous forme de machines, processus, sous–application ou objet non strict) n'apparaissent dans l'interface (sinon on va à l'encontre de l'approche "protection du contenu" propre à l'approche orientée par les objets). *La seule exception concerne les entités composant une collections qui sont par définition visibles.*

Plus précisément, la forme générale de l'interface d'une entité (sauf collection) est celle–ci :

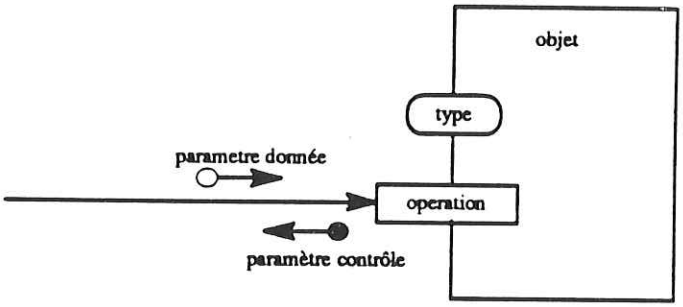


Figure 4.1. interface d'entité

La boîte principale représente une entité et décrit donc un objet ou un modèle d'objets. Les éléments à cheval sur le bord de la boîte représentent ce qui est visible de l'extérieur, c'est–à–dire l'interface de l'entité.

La description du corps d'une entité composite (sous–application, objet non strict ou collection) se fait par un diagramme dans lequel apparaissent les composants de l'entité et leurs relations.

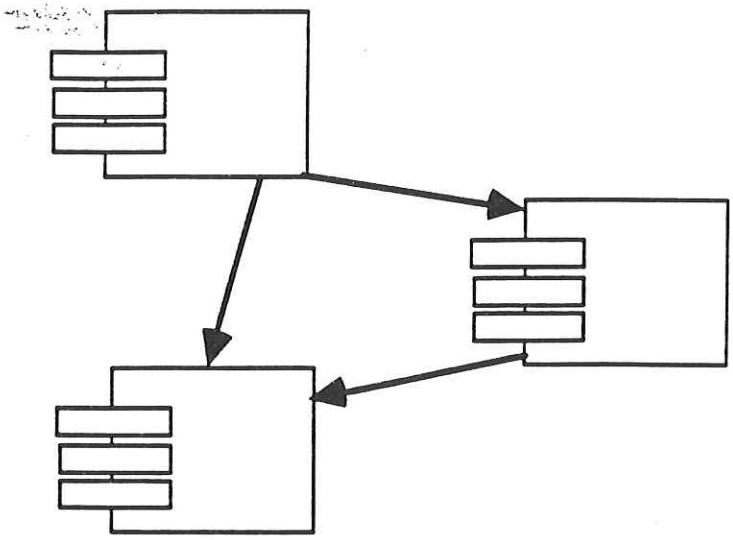


Figure 4.2. Corps d'entité composite

La description du corps d'une entité de base se fait en général en conception détaillée, en utilisant un langage de description adapté (type "PDL", Program Design Language).

Dans une conception, le concepteur va réutiliser des entités (NDS – Non Developmental Software) : il est important de reconnaître ces entités, car elles ne seront probablement pas détaillées dans la conception. Elles sont représentées avec des doubles barres latérales, comme le montre la figure 4.3.

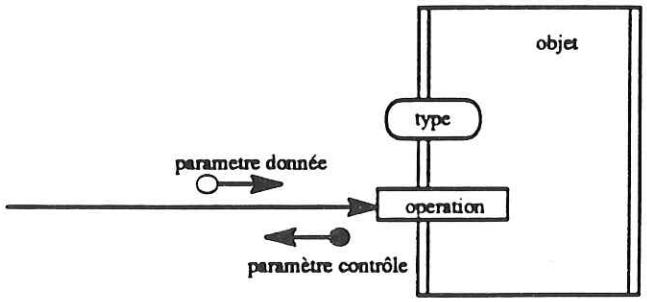


Figure 4.3. Interface d'entité réutilisée

4.1.3. Généralités sur le lien entre les entités et les CSCs/CSUs

Vu du document de conception (SDD) :

- le CSU est une feuille terminale dans la décomposition pour la conception.
- le CSC est un assemblage de sous–composants (CSCs ou CSUs) qui n'a pas de contenu en dehors de celui de ses composants.

En termes de composants :

> le CSU est assimilable en première approche à une unité de programmation objet constituée d'un fichier "spécification" et d'un fichier "corps".

Le principe général est qu'une entité composite (sous–application, objet non strict et collection) correspond donc à un CSC, et qu'une entité de base (machine, processus, types abstraits et génériques) correspondent à des CSUs.

Les particularités ou exceptions à ce principe seront exposées dans les chapitres propre à chaque entité.

Note : pour le diagramme d'environnnement d'une entité, il est possible de préciser au niveau OOSD si le lien concerne la spécification ou le corps de l'entité, en utilisant une annotation.

4.1.4. Généralités sur l'outil

L'outil se compose de plusieurs éditeurs :

- TOOD/C Design Editor : il permet de réaliser les différents diagrammes;
- Object Annotation Editor : il permet d'associer du texte aux diagrammes et à ses éléments constituants.

Tout ce qui est créé par ces éditeurs peut être inclus dans le dictionnaire.

L'outil permet en particulier de :

- vérifier que la conception est complète et respecte certaines règles;
- générer des documents de façon automatique.

Un concept particulièrement important est la notion de *définition* et de *réutilisation* d'une entité : une entité est *définie* lorsqu'on lui a associé une annotation. Une annotation est un ensemble de notes qui peut avoir une structure assez complexe.

Lorsqu'on génère le dictionnaire, l'entité annotée, donc définie, devient candidate à la réutilisation. Par le mécanisme de la commande *reuse*, on peut faire figurer l'entité dans un autre diagramme.

La commande *reuse* permet de sélectionner dans un catalogue les entités qu'on peut utiliser dans un diagramme, ainsi que tout ou partie de leur interface.

4.1.5. Généralités sur la traduction en C ANSI

4.1.5.1. Le choix Ansi

La première version du langage C définit par Kerninghan et Ritchie présentait des lacunes au niveau des concepts et notamment des contrôles de types.

Un certain nombre d'implémenteurs du langage ont ajouté des concepts utiles, tel que les énumératifs, les prototypes, les manipulations sur les structures.

La normalisation du langage C regroupe la plus part de ces innovations pour en faire un langage correct et industriel.

4.1.5.2. Les innovations du langage C ansi

Pointeur universel

Le pointeur universel void* permet de réaliser des fonctions indépendantes du type de l'objet manipulé. Les opérations d'incrémentation et de décrémentation du pointeur sont proscrites sur un pointeur void*, ainsi des opérations malvenues sont évitées.

Contrôle de paramètres

Les paramètres déclarés dans un sous–programme peuvent être vérifiés lors de l'appel au sous–programme. Le nombre et les types des paramètres font l'objet d'une vérification par le compilateur.

Le prototype permet de définir une spécification de sous–programme. L'utilisation systématique des prototypes permet de vérifier que les appels sont en concordances avec les sous–programmes appelés.

Le paramètre unique void permet de distinguer les sous–programmes sans paramètre et les déclarations sans vérification.

type fonction (void); permet d'indiquer que la fonction n'a pas de paramètre
par contre
type fonction (); permet d'indiquer que les vérifications sur les paramètres ne doivent pas être effectuées.

Les constantes

Les constantes introduites pas const permettent de s'affranchir de l'utilisation de "define". Les identificateurs introduits par ce dernier n'étant pas reconnus pas les débogueurs symboliques.

L'utilisation de const dans la déclaration des paramètres formels permet de séparer les genres "in" et "in out" sur ces derniers. Ainsi il devient possible de passer un tableau sans donner la possibilité de modification des éléments du tableau. La même chose est possible pour les structures, les structures pouvant être passées par adresse par l'implémenteur sans être déclarées pour autant par des pointeurs.

Les énumératifs

Les énumératifs existaient dans la plupart des versions du langage C, ils deviennent réalité.
Là encore ils permettent la déclaration de constantes entières et l'utilisation fastidieuse de "define" pour palier à son absence.

Normalisation des conversions entières

Les conversions entre numériques sont désormais normalisées. Les opérations ne se font plus en double précision systématiquement ce qui alourdissaient inutilement le code généré. L'utilisateur maîtrise mieux le comportement de ses algorithmes numériques

Les opérations sur les structures

Les affectations entre deux structures sont maintenant possibles. Les fonctions peuvent maintenant restituer une donnée structurée.

La taille des identificateurs

La taille des identificateurs n'est plus limitée. Cependant, la limitation reste dependante du choix des produits utilisés. La limite à 32 caractères significatifs des identificateurs internes n'est en soit pas une contrainte très forte, par contre la limite à 8 caractères reste insuffisante pour satisfaire le besoin d'identificateurs externes significatifs dans les programmes. Cette dernière est liée aux chaînes en aval (assembleurs et éditeurs de liens).

4.1.5.3. Passage à la normalisation

Le code écrit en C non normalisé est récupérable sans changement en C *ansi. Des problèmes peuvent apparaître dans le cas des numériques mal utilisés en C traditionnel.
Les "include" utilisateur en langage C traditionnel ne permettent pas d'utiliser les possibilités de contrôle sur les types. Une réécriture peut s'avérer souhaitable pour garder cette possibilité, mais non nécessaire si le projet peut s'absoudre d'un certain nombre de contrôles. Les "mauvaises utilisations" des éléments de la bibliothèque standard seront détectées.
Dans le cas d'interface avec le système unix, quelques modifications seront nécessaires au niveau des retours de fonctions dans les entrées et sorties. La charge est légère et peut se faire de manière automatique par un petit outil de coût de réalisation modique.

4.1.5.4. Avenir

De plus en plus les compilateurs C sont normalisés. Les choix proposés par la normalisation sont souvent pertinents et proviennent de remarques des utilisateurs.
Les constructeurs de machines proposent le C *ansi en plus du C traditionnel.
Le C *ansi est une bonne passerelle entre le C traditionnel et le C++. Les programmes écrits en C *ansi sont récupérables dans une chaîne de production C++.

4.1.5.5. Les entités de la conception

Le principe repose sur les notions de spécification et de corps d'une entité. Les concepts d'"include", de "prototype" sont les supports de l'implémentation des entités.

Une entité sera constituée de deux fichiers, un premier fichier (XXX.h) utilisé en "include" qui traduira la notion de spécification, et un second fichier (XXX.c) qui traduira la notion de corps.

La visibilité sur la spécification sera traduite par la commande "include" dans le module qui souhaite obtenir une visibilité sur cette entité.

La visibilité d'une spécification par son corps sera traduite aussi par la commande "include", cela permettant d'assurer la cohérence entre utilisation et définition.

Le fichier "spécification" contiendra la description de toutes les parties visibles de l'entité, le fichier "corps" traduira les parties non visibles de l'entité et nécessaires à l'implémentation.

4.2. La machine

4.2.1. Introduction

C'est l'entité dédié à la description d'un objet de base, passif.

Une machine contient un ensemble d'informations (variables de types abstraits) qu'il gère sur demande d'un processus ou d'une autre machine.

La gestion de ces informations est réalisée par des opérations, les unes externes, donc accessibles par l'environnement, les autres internes, définies pour les besoins propre de gestion de la machine. Une machine réalise donc les traitements correspondant aux opérations sur ses ressources, sur demande d'un processus ou d'une autre machine.

Alors que les types abstraits sont des modèles de données (cf AUCUN LIEN), les machines contiennent effectivement des données. Elles "existent", au sens où elles ont une représentation en mémoire. Une machine peut également représenter une abstraction d'une ressource matérielle.

Une machine est de tendance statique puisqu'elle correspond à un stockage d'information. Elle n'est pas créée ni supprimée dynamiquement[3]. Elle contient une information rémanente. Quand une entité maître appelle une opération d'une machine, cette machine exploite l'information qu'elle contient et qui est inconnue par l'appelant.

Une machine doit s'assurer que l'enchaînement des demandes d'opérations effectuées par l'environnement (ses entités "maîtres") est cohérent. Elle ne peut agir sur ceux qui emmettent ces demandes, mais elle peut vérifier que ce qu'on lui demande de toute part (car elle peut avoir plusieurs "maîtres") est possible à réaliser et que ces demandes ne vont pas la mettre dans un état (interne) incohérent.

En particulier, l'une des responsabilités de la machine, responsabilité qui peut s'avérer délicate et critique, est de gérer l'accès concurrentiel à ses données via les opérations qu'elle met à disposition de ses entités maîtres. En effet, quand deux processus appellent les opérations d'une machine, c'est sur les mêmes données (celles de la machine) que ces opérations parallèles s'exécutent!

Comme un processus, une machine peut être maître ou esclave de la communication avec son environnement, selon qu'elle appelle les opérations d'entités de l'environnement ou que ses opérations sont appelées par l'environnement.

Pour réduire les problèmes de blocages, *une machine ne peut appeler que des opérations d'autres machines* (pas d'appel d'opération de processus). Les dérogations doivent être dûment motivées et signalées.

Comme un processus, la machine est vue de l'extérieur comme une boîte noire comprenant un ensemble d'opérations (et les exceptions associées).

3. De plus, la "position", c'est-à-dire l'adresse, de ses données dans la mémoire est fixe.

4.2.2. Interface

Une *machine* est une entité dont le but est de rendre un certain nombre de services à ses maîtres. Vu de l'environnement, elle est donc caractérisée par les services qu'elle rend et les services qu'elle demande à d'autres entités. La définition de l'interface d'une machine contient donc :

- un ensemble d'opérations (dites *externes*), correspondant aux services que peut rendre la machine, avec la définition de leurs paramètres formels;
- des déclarations (de noms) d'exceptions qui nomment les cas de fonctionnement exceptionnels (ou de disfonctionnement) des opérations.

Représentation sous l'outil OOSD/TOODC :

L'interface d'une machine est définie sur un diagramme qui a le même nom.
Utiliser le symbole paquetage.
Tous les composants de ce diagramme sont annotés.
Les paramètres des opérations sont renseignés.
Les annotations d'une opération pourront être reportées si on ouvre un nouveau diagramme pour cette opération.

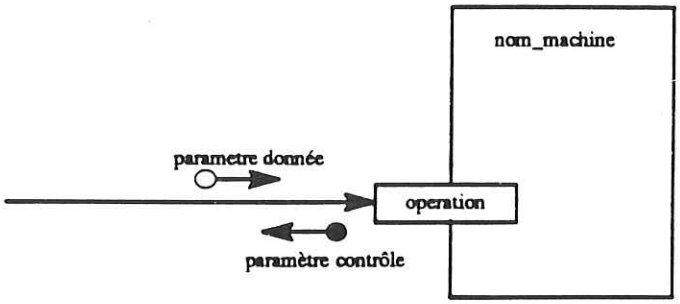


Figure 4.4. modèle d'interface de machine

Annotation obligatoire du diagramme :

Diagram : *nom_machine*
Name : *nom_machine*
Author : *nom_du_concepteur*
Generated : *date*

Annotation obligatoire de la machine :

paquetage : *nom_machine*
Name : *nom_machine*
Note Description *texte de description*
Requirement : *identificateur local de l'exigence*
RequirementName : *identificateur officiel de l'exigence*
RequirementDoc : *référence à la liste des documents cités en référence : SRS, IRS ou SCNxxx*
Constraint : *mot_clé identifiant la contrainte, par exemple : mémoire*
Note Description *éditer un texte de description de la contrainte*

Annotation des opérations : *même chose que la machine.*

Le symbole ←○ représente une donnée (objet instance d'une classe), à laquelle sont associés un nom et son type.

Le symbole représente une donnée de nature contrôle, c'est–à–dire que cette donnée sert uniquement à modifier le comportement de l'objet représenté par l'entité qui reçoit cette donnée.

Dans certain cas, on peut trouver sur l'interface un type de données (genre type énumératif par exemple) quand il sert uniquement à la communication avec cette machine.

Exemple d'interface de machine:

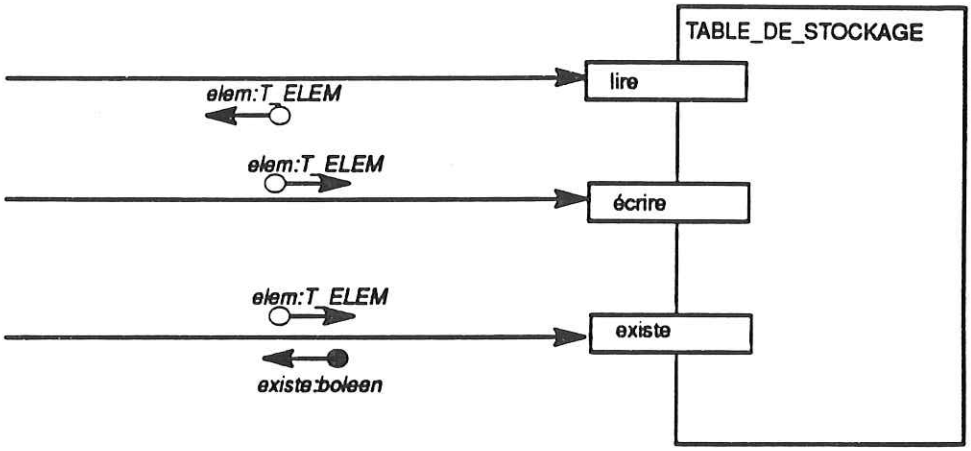


Figure 4.5. Exemple d'interface de machine

Annotation du diagramme :
Diagram : *table de stockage*
Name : *table de stockage*
Author : *dupont*
Generated : *11:21:15 1 24 1991*

Annotation obligatoire de la machine :
paquetage : *table de stockage*
Name : *table de stockage*
Note Description ...
Requirement : 1.1.3
RequirementName : 1.1.3
RequirementDoc : *SRS_Ind*
Constraint : ...
....

4.2.3. Environnement

Il définit les composants utilisés par la machine.

- Si ces composants sont des machines internes de la même sous–application, on présente ces machines avec une interface réduite à ce qui est utilisé par la machine.
- Si ce sont des types abstraits d'une collection, on les présente en préfixant leur nom par le nom de la collection et avec une interface réduite à ce qui est utilisé par la machine.
- Si les opérations dont a besoin la machine sont définies dans d'autres sous–applications, on présente ces sous–applications avec une interface réduite à ce qui est utilisé par la machine.

L'environnement d'une machine *nom_de_machine* est défini sur un diagramme dont le nom est *nom_de_machine*_env.
Le diagramme seul est annoté.

Création du diagramme d'environnement :
Dans le menu, sélectionner la commande *options.push_env* puis cliquer sur l'entité.
Le diagramme d'environnement est chargé. S'il n'existe pas, il est créé (diagramme vide).

Pour remplir le diagramme, insérer les symboles des entités, appeler les interfaces par la commande *reuse*.

Générer le dictionnaire.

Accès au diagramme d'environnement :
Dans le menu, sélectionner la commande *options.push_env* puis cliquer sur l'entité.
Ou bien appeler le menu associé au champ *Diagram* du *Control Panel* et choisir le diagramme.

Exemple :

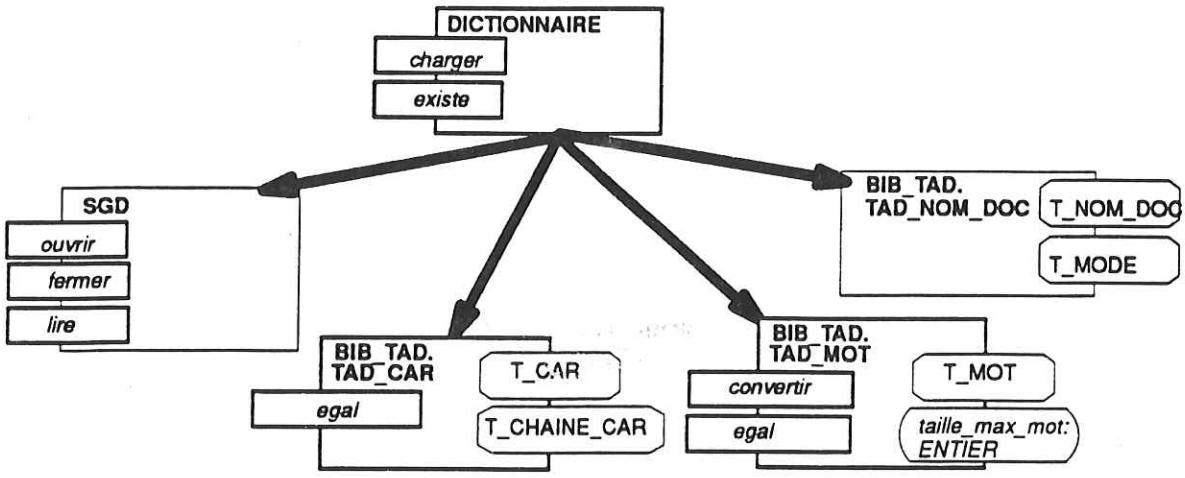


Figure 4.6. Environnement de la machine DICTIONNAIRE

4.2.4. Corps

Définir le corps d'une machine correspond à définir:

- les ressources (ressources matérielles ou informations logiques sous forme de variables et/ou de constantes) gérées par la machine,
- les opérations internes éventuelles (non accessibles par l'environnement, mais utiles pour la réalisation des opérations externes.
- comment les opérations (externes et internes) sont définies, c'est à dire leur algorithmique.

La description du corps d'une machine est du domaine de la conception détaillée. Le principe général est que la conception détaillée est décrite dans les fichiers contenant le code source sous la forme d'un PDL, afin de permettre une meilleure cohérence entre le code source et la conception détaillée, en particulier lors de modifications.

Cependant, si le concepteur le juge utile, le corps d'une machine peut être représenté graphiquement, comme la figure ci–après le montre.

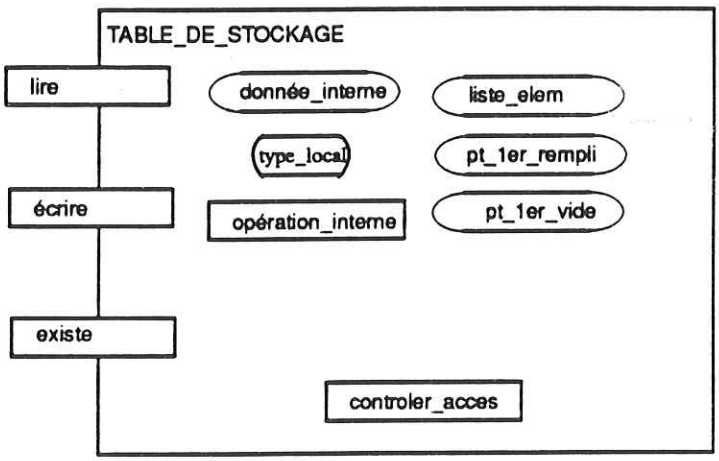


Figure 4.7. Exemple de représentation graphique d'un corps de machine

Dans le cas où le corps de la machine est repésenté graphiquement :

Le corps d'une machine *nom_de_machine* est défini sur un diagramme dont le nom est *nom_de_machine*_body.
Seules les données et opérations internes sont annotées.
Les annotations d'une opération pourront être reportées si on ouvre un nouveau diagramme pour cette opération (a priori rare).

Création du diagramme de corps :
Dans le menu sélectionner la commande *options.push_body* puis cliquer sur l'entité.
Le diagramme de corps est chargé. S'il n'existe pas, il est créé (diagramme vide).

Pour remplir le diagramme, insérer le symbole de l'entité, appeler l'interface par la commande *reuse*.
Ajouter les éléments internes et les annoter.
Générer le dictionnaire.

Accès au diagramme de corps :
Dans le menu sélectionner la commande *options.push_body* puis cliquer sur l'entité.Ou bien appeler le menu associé au champ *Diagram* du *Control Panel* et choisir le diagramme.

4.2.5. Machines dégénérées

Une machine dégénérée est une machine dans laquelle des données apparaissent dans l'interface de la machine et sont donc visibles (et utilisables directement) par l'environnement. Elles introduisent alors des données partagées, ce qui introduit un risque souvent grave dans le développement.

Comme leur nom l'indique, elles ne doivent être utilisées que dans des cas rares, et leur utilisation doit être fortement justifiée.

Exemple d'utilisation : des données partagées entre deux processus situés sur des processeurs différents, qui doivent se situer dans une mémoire partagée qui ne peut contenir d'exécutable dans le cas ou le compilateur n'accepte pas les clauses d'adresses adéquates.

4.2.6. Traduction en C ANSI

La machine se comporte comme un type abstrait de données au niveau de l'organisation aux exceptions suivantes.

1. Les types

   Le fichier de "spécification" ne propose pas de types sauf ceux nécessaires pour utiliser les opérations de la machines (e.g un type énumératif0.

2. Les données de la machine

   Les données de la machine sont déclarées dans le fichier "corps" et sont classées "static":

   ```
   static int   etatReacteur;
   ```

   Dans le cas de la "machine dégénérée", les données qui la rendent dégénérée seront déclarées comme les constantes visibles:

   Dans la "spécification"

   ```
   extern      int    reacteur_etatReacteur;
   ```

   Dans le "corps"

   ```
   int    reacteur_etatReacteur;
   ```

3. Les opérations

   Il n'y a pas de contraintes sur le premier paramètre.

   Dans la "spécification"

   ```
   void  reacteur_remplir ( int );
   ```

   Dans le "corps"

   ```
   void  reacteur_remplir ( int )
   { ... }
   ```

4.3. Processus

Ce chapître reste à préciser, les paragraphes suivants proposent les concepts utiles pour le concepteur, cependant leur implémentation en C reste dépendant de l'environnement d'execution.

4.3.1. Introduction

Le processus est l'une des entités qui permet l'expression du parallélisme. A la différence des entités représentant des objets composites qui peuvent être "multi–*Thread–of–control*[4]", le processus ne contient qu'un "*Thread–of–control*".

Un processus réalise un traitement séquentiel qui s'effectue parallèlement aux traitements associés à d'autres processus.

Plusieurs processus d'une même application peuvent se synchroniser en échangeant des signaux ou des messages, soit de manière synchrone (technique du rendez–vous), soit de manière asynchrone (boîte_à_lettre, Queue ...).

Un processus se synchronise en direct avec l'extérieur tantôt en tant que maître, tantôt en tant qu'esclave.

- lorsqu'il est maître de la synchronisation, c'est lui qui demande à échanger des informations avec un autre processus; il appelle pour cela une opération d'échange définie dans un autre processus;
- lorsqu'il est esclave de la synchronisation, il se met en attente d'un échange d'information avec un processus qu'il ne connaît pas et qui sera le maître de l'échange. Pour permettre au processus maître de faire cet échange, il doit donc définir les opérations d'échange.

Vu de l'extérieur, le processus est une boîte noire comprenant des opérations de communication et les paramètres de la communication.

Chaque processus appartient à une sous–application.

Pour des raisons de maîtrise du parallélisme, il faut si possible éviter de créer des processus dynamiquement.

4.3.2. Interface

Un processus fournit des opérations qui permettent à son environnement d'échanger des messages et de se synchroniser.

4. Il n'existe pas de terme français équivalent (fil de contrôle). Cela correspond à l'ensemble des chemins que peut suivre un "jeton de contrôle".

Représentation sous l'outil OOSD/TOODC Design Editor :

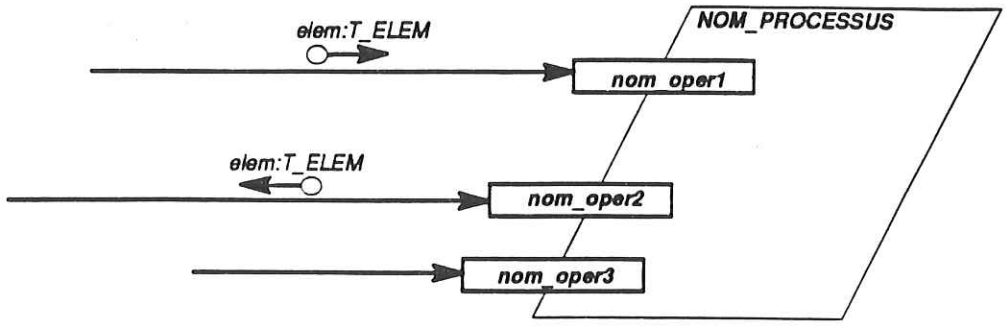


Figure 4.8. Modèle de l'interface d'un processus

Note : cette manière de mettre une "embase" à une tache n'est pas nécessaire en conception mais est une contrainte d'interprétation de l'outil.

L'interface d'un processus est définie sur un diagramme qui a le même nom.
Utiliser le symbole de tâche.

Tous les composants de ce diagramme sont annotés.
Les paramètres des opérations sont renseignés.
Les annotations d'une opération pourront être reportées si on ouvre un nouveau diagramme pour cette opération.

Annotation obligatoire du diagramme :

Diagram : gestion buff
Name : gestion buff
Author : dupont
Generated : 11:21:15 1 24 1991

Annotation obligatoire du processus :

Process : gestion buff
Name : gestion buff
Note Description *(éditer un texte de description)*
Requirement : 2.3.2
RequirementName : 2.3.2
RequirementDoc : SRS

Annotation des opérations : *même chose que processus.*

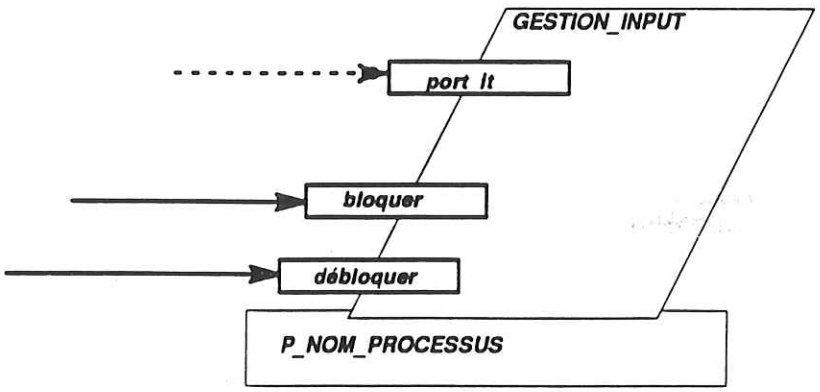


Figure 4.9. Exemple de l'interface d'un processus

Pour renseigner sur les contraintes associées à une opération (exemple : l'opération *agir*) on prend les conventions suivantes :

- Opération à haut degré de synchronisation (timing analogue à l'appel de procédure) : opération *agir*,

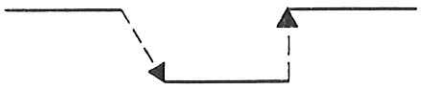


Figure 4.10. Opération à haut degré de synchronisation

lien associé : *Formal Task Entry Call.*

- Opération à bas degré de synchronisation :
opération *sync_agir*,

Figure 4.11. Opération à bas degré de synchronisation

lien associé : *Formal Task Entry Call.*

- Opération asynchrone :

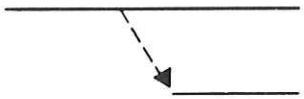


Figure 4.12. Opération asynchrone

opération *ev_agir* (notion d'évènement), *fifo_agir* (mise en file d'attente des évènements) :

lien associé : *Formal Interrupt.*

Remarque : il est aussi possible de faire apparaître la communication par une machine (instance d'une machine générique) représentant le mécanime de communication : machine_événement, machine_fifo, ... Les deux processus se branchent alors sur la machine.

opération *it_agir* (interruption hardware) :

lien associé : *Interrupt.*

- Opération à spécification de time–out (l'opération doit être exécutée à l'intérieur d'un certain intervalle de temps) :

utiliser un paramètre contrôle pour spécifier le time–out.

Un processus cyclique est représenté comme ayant une opération externe nommée TEMPO (nom réservé).

4.3.3. Environnement

Comme celui d'une machine (cf. 4.2.3.)

L'environnement d'un processus *nom_de_processus* est défini sur un diagramme dont le nom est *nom_de_processus*_env.
Le diagramme seul est annoté.

Création du diagramme d'environnement :
Dans le menu, sélectionner la commande *options.push_env* puis cliquer sur l'entité.
Le diagramme d'environnement est chargé. S'il n'existe pas, il est créé (diagramme vide).

Pour remplir le diagramme, insérer les symboles des entités, appeler les interfaces par la commande *reuse*.
Générer le dictionnaire.

Accès au diagramme d'environnement :
Dans le menu, sélectionner la commande *options.push_env* puis cliquer sur l'entité.
Ou bien appeler le menu associé au champ *Diagram* du *Control Panel* et choisir le diagramme.

4.3.4. Corps

Globalement comme une machine (cf 4.2.4.).

Le corps d'un processus *nom_de_processus* est défini sur un diagramme dont le nom est *nom_de_processus*_body.
Seules les données et opérations internes sont annotées.
Les annotations d'une opération pourront être reportées si on ouvre un nouveau diagramme pour cette opération.

Création du diagramme de corps :
Dans le menu, sélectionner la commande *options.push_body* puis cliquer sur l'entité.
Le diagramme de corps est chargé. S'il n'existe pas, il est créé (diagramme vide).

Pour remplir le diagramme, insérer le symboles de l'entité, appeler l'interface par la commande *reuse*.
Ajouter les éléments internes et les annoter.
Générer le dictionnaire.

Accès au diagramme de corps :
Dans le menu, sélectionner la commande *options.push_body* puis cliquer sur l'entité. Ou bien appeler le menu associé au champ *Diagram* du *Control Panel* et choisir le diagramme.

Un processus pilote une certaine quantité d'activités. Les opérations internes à un processus correspondent au découpage de ce pilotage. Elles peuvent être représentées dans le symbole de tache associé au processus.

4.3.5. Traduction en C ANSI

L'implémentation est à définir en fonction de l'environnement d'exécution disponible .

4.4. Sous–application

4.4.1. Introduction

Lorsqu'une application temps réel est importante, il est nécessaire :

- de pouvoir la décomposer en sous–applications,
- d'exprimer les relations (d'utilisation, de communication ...) entre ces sous–applications.

De plus, il faut pouvoir itérer ce procédé de découpe jusqu'à atteindre des sous–applications que le concepteur peut décrire en termes d'entités effectives que sont les processus et les machines abstraites (seules entités effectives).

C'est le concept de *sous–application* qui répond à ce besoin. Une sous–application décrit une "architecture" parallèle (c'est à dire contenant des processus) ou séquentielle.

C'est donc une entité décomposable. En fait, c'est un assemblage d'entités et seulement un assemblage d'entités. Cela veut dire que la sous–application, hors des entités qu'elle contient ne représente rien.

Une sous–application masque son organisation interne (principe de base de la conception orientée objets).

Une sous–application peut être de forme stricte (elle est simplement nommée sous–application) ou de forme non stricte (elle est nommée sous–application non stricte).

Alors que la sous–application (stricte) n'est qu'une enveloppe, la sous–application non stricte contient des informations ou des opérations qui n'appartiennent à aucun des composants. Son domaine d'utilisation est le même que celui de la sous–application, mais il correspond à une approche conceptuelle très différente :

- dans le cas de la sous–application, un tout n'est que l'ensemble de ses parties,
- dans le cas de la sous–application non stricte, un tout est l'ensemble de ses parties plus "du liant".

La forme non stricte de sous–application présente un certain nombre de risques et son utilisation doit donc être contrôlée[5] :

- introduction de données dans la sous–application non stricte qui n'appartiennent à aucun des composants : cela revient à introduire des données partiellement globales, avec tous les risques associés (accès concurrentiel très mal contrôlé, impact d'une modification sur une opération qui utilise une telle donnée sur les autres opérations qui utilisent cette donnée, impact de la modification de la donnée elle–même sur toutes les procédures qui l'utilisent ...); c'est le risque le plus grave;

5. Dans certains cas, cette approche paraît la plus satisfaisante et la plus naturelle. Mais il faut savoir que les objets non stricts sont "traductibles" en sous–applications soit en "déléguant" la propriété d'une information ou la responsabilité du pilotage à l'un des composants, soit en identifiant une entité correspondant à ce qui n'appartient à aucun des autres composants.

- introduction d'opérations dans la sous–application non stricte, qui n'appartiennent à aucun des composants : cela revient à introduire des "pilotes locaux" qui sont essentiellement des ordonnanceurs d'appels des opérations des composants internes (s'ils sont plus que cela, et ils peuvent l'être, le risque est encore plus grand); l'expérience montre que les développeurs ne réalisent pas toujours les test unitaires sur ces opérations avec le sérieux requis car elles leur paraissent "évidemment tourner" et attendent l'intégration pour tester ces opérations.
- l'interprétation en termes de CSCs et de CSUs est là encore délicate : en effet, une sous–application non stricte est à la fois CSC (puisqu'il se décompose en sous–composants) et un CSU (puisqu'il contient du code propre). Une proposition possible est qu'il se traduit en CSC de même nom que la sous–application non stricte, plus un CSU de nom "global_nom_d_objet_non_strict"[6].

Une sous–application peut se décomposer :

- en sous–applications (et ce procédé de décomposition peut être itéré autant de fois qu'il est nécessaire) de forme stricte ou non stricte,
- en processus,
- en machines,

et peut contenir

- une ou plusieurs collections si celles–ci lui sont locales,
- directement un type abstrait (sans passer par une collection) si c'est le seul type abstrait ou un générique dans les mêmes conditions.

Conseil : il faut éviter de décomposer une sous–application de manière non homogène, c'est–à–dire de mélanger des entités de base avec des entités composites. En effet, leur interprétation en terme de CSCs et de CSUs est différente et peut poser des problèmes lors de la génération du document de conception SDD : le CSC sous–application apparaîtra d'une part comme décomposé en CSCs (correspondants aux entités composites) dans le chapitre 3 du SDD et d'autre part comme décomposé en CSUs (correspondants aux entités de base) dans le chapitre 4.

Une sous–application peut être active (c'est–à–dire contenir au moins un processus) ou passive (ne pas contenir de processus).

Remarque : la différence marquée entre la machine et la sous–application n'est pas associée à un point de vue "objet", ni au fait que l'utilisation d'un objet décrit par une sous–application est fondamentalement différente de celle d'un objet décrit par une machine, mais à des considérations basées sur le risque (complexité structurelle, possibilité de "multi–parallélisme", ...) ou la documentation associée et son lien avec les concepts de CSC et CSU de la norme DOD–STD–2167A.

4.4.2. Interface

Décrire l'interface d'une sous–application consiste à décrire la manière de communiquer du sous–application avec son environnement.

6. Cette solution est finalement quasiment la même chose que la deuxième proposition de traduction en sous–application de la remarque 5 ci–dessus, vis à vis de la DOD–STD–2167A/SDD, mais pas vis à vis du code. Ce CSU n'est alors qu'un regroupement pour des besoins DOD–STD–2167A : c'est un CSU "fourre–tout".

L'interface comprend également deux indications :

- la première pour indiquer si la forme (stricte ou non stricte) de la sous–application,
- la seconde pour indiquer si elle est active ou passive.

Ces deux indications sont représentées par un "commentaire" associé au nom de la sous–application :

- "(SA)" pour une sous–application stricte active,
- "(SAP)" pour une sous–application stricte passive,
- "(NS)" pour une sous–application non stricte active,
- "(NSP)" pour une sous–application non stricte passive,

Notons que ce commentaire permet également de différencier une sous–application d'une machine.

Bien que ce commentaire ne soit pas de nature "interface" (mais caractérise le "comment" de l'entité), il donne une indication sur la complexité et donc le risque associé à l'entité. Par exemple, une entité utilisée par la sous–application ne sera pas de même nature (protégée ou non protégée) suivant que la sous–application est on n'est pas passive.

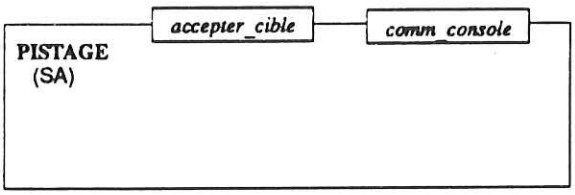


Figure 4.13. Interface d'une sous–application

L'interface d'une sous–application est définie sur un diagramme qui a le même nom.
Utiliser le symbole paquetage
Créer un commentaire associé au nom du paquetage (SA, SAP, NS ou NSP)
Tous les composants de ce diagramme sont annotés.

Annotation obligatoire du diagramme :

Diagram : *nom_sous–application*
Name : *nom_sous–application*
Author : *nom_du_concepteur*
Generated : *date*

Annotation obligatoire de la sous–application :

paquetage : *nom_sous–application*
Name : *nom_sous–application*
Note Description *texte de description*
Requirement : *identificateur local de l'exigence*
RequirementName : *identificateur officiel de l'exigence*
RequirementDoc : *référence à la liste des documents cités en référence : SRS, IRS ou SCNxxx*
Constraint : *mot_clé identifiant la contrainte, par exemple : mémoire*
Note Description *éditer un texte de description de la contrainte*

Annotation des opérations : *même chose que la sous–application.*

4.4.3. Environnement

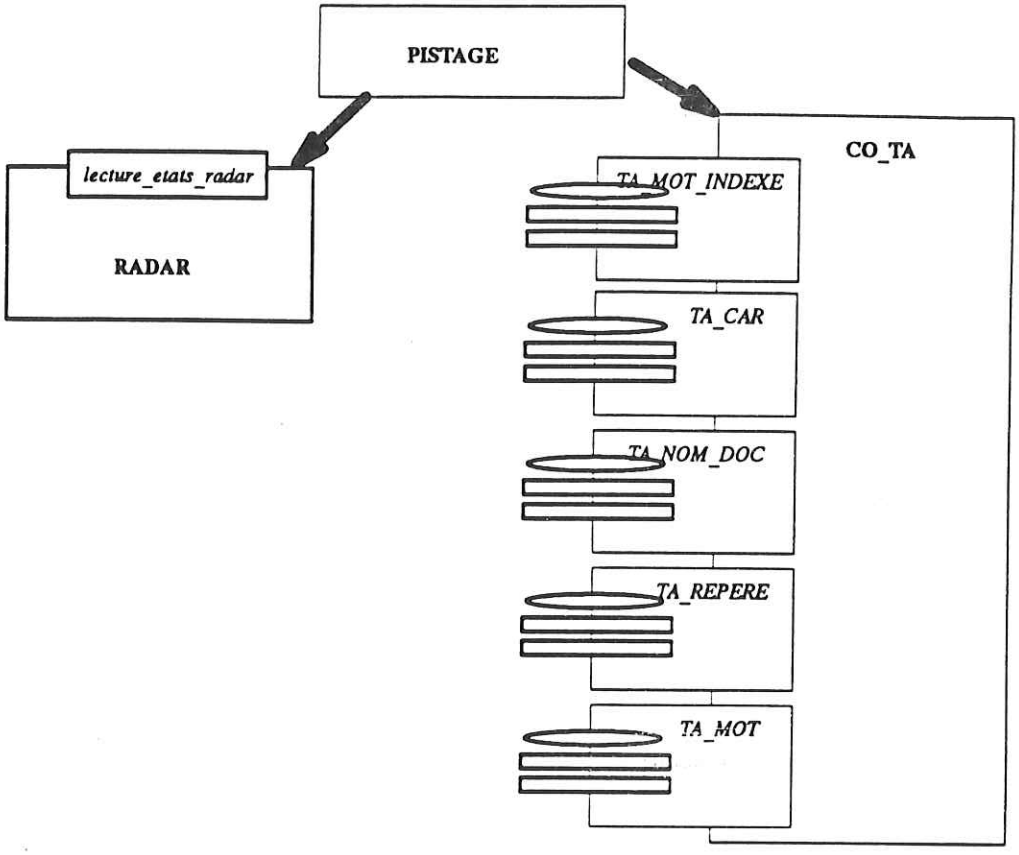


Figure 4.14. Environnement d'une sous–application

L'environnement d'une sous–application *nom_de_sous–application* est défini sur un diagramme dont le nom est *nom_de_sa_*env.
Le diagramme seul est annoté.

Création du diagramme d'environnement :
Dans le menu, sélectionner la commande *options.push_env* puis cliquer sur l'entité.
Le diagramme d'environnement est chargé. S'il n'existe pas, il est créé (diagramme vide).
Pour remplir le diagramme, insérer les symboles des entités, appeler les interfaces par la commande *reuse*.
Générer le dictionnaire.

Accès au diagramme d'environnement :
Dans le menu, sélectionner la commande *options.push_env* puis cliquer sur l'entité.
Ou bien appeler le menu associé au champ *Diagram* du *Control Panel* et choisir le diagramme.

4.4.4. Corps

C'est au niveau du corps de la sous–application que va être marquée la différence entre sous–applications strictes et non strictes. En effet les données propres à la sous–application non stricte apparaîtront, les opérations de l'interface pourront être liées à (car elles appelent) plusieurs opérations de composants, ou au contraire à aucune opération de composant.

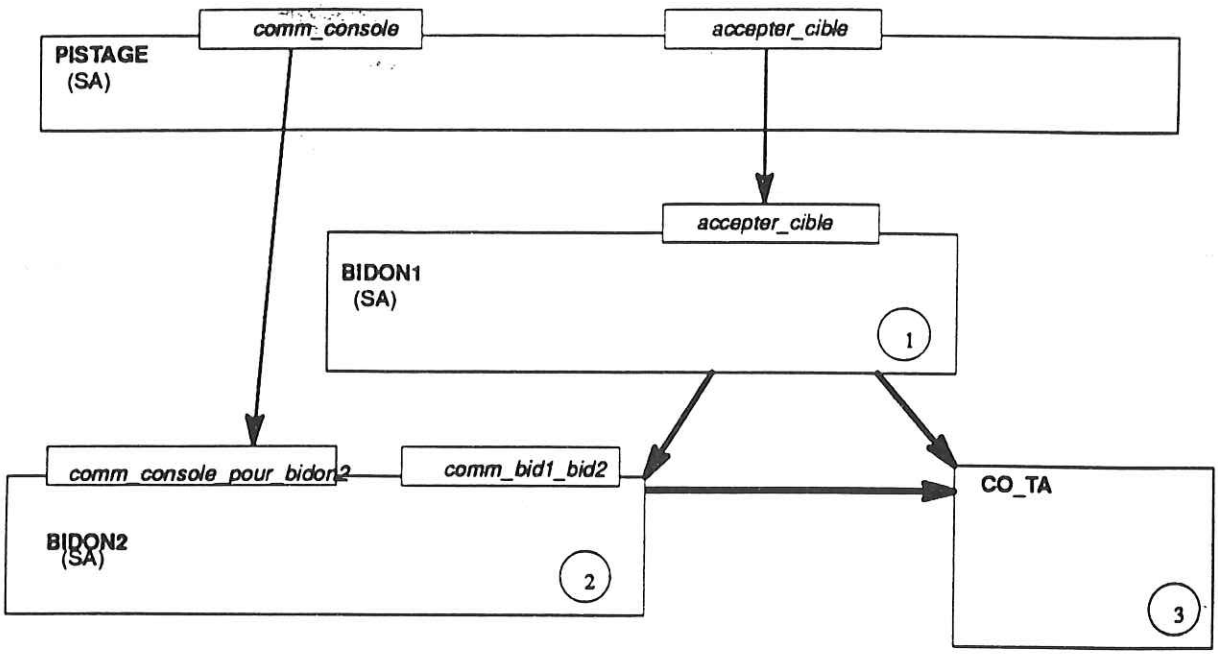


Figure 4.15. Exemple de corps de sous–application se décomposant en sous–applications

Si une opération externe d'une sous–application doit être reliée à une opération de processus, il faudrait pouvoir le voir (cas du rendez–vous ou d'une opération de synchronisation) : pour cela, utiliser un préfixe pour l'opération (du fait que l'outil interdit des entrées sur le symbole de paquetage). Cela permet d'indiquer qu'il s'agit d'une opération particulière (synchronisation).

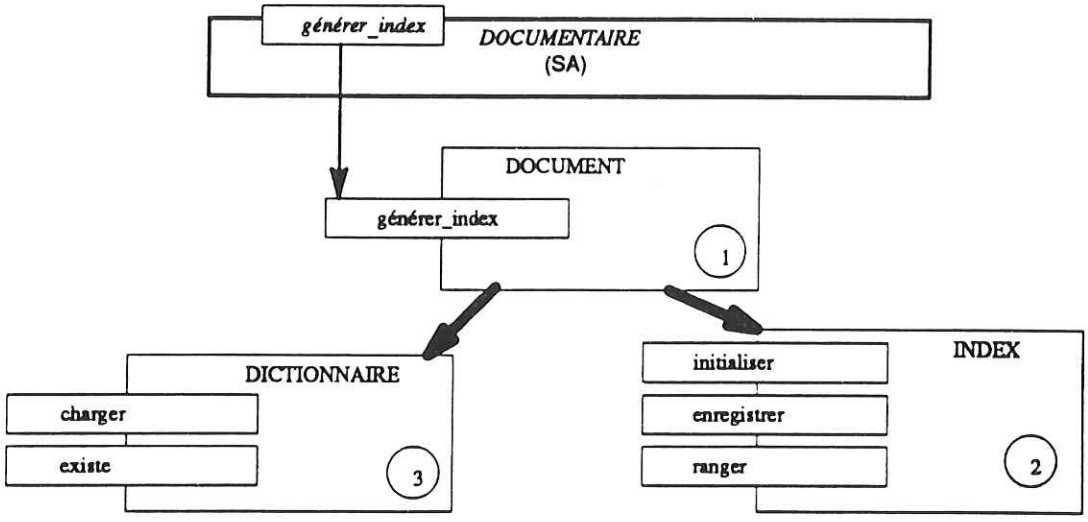


Figure 4.16. Exemple de corps de sous–application se décomposant en machines

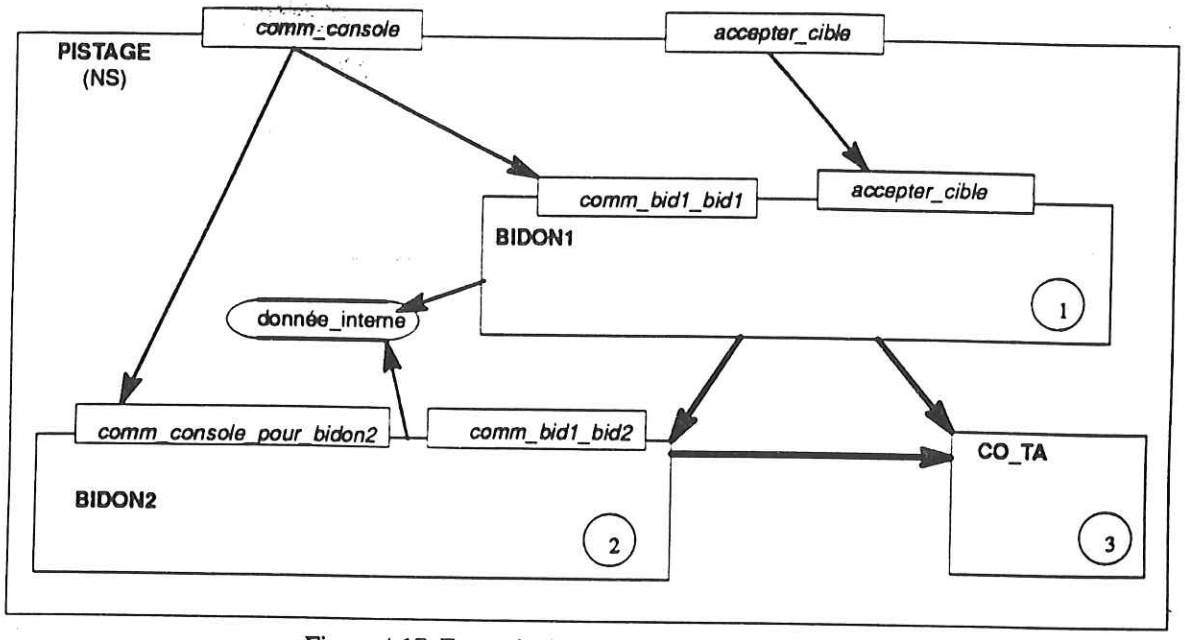


Figure 4.17. Exemple de corps de sous–application non stricte.

Le corps d'une sous–application *nom_de_sous–application* est défini sur un diagramme dont le nom est *nom_de_sous–application*_body.
Ses composants ne sont pas annotés puisqu'ils sont définis dans d'autres diagrammes.

Création du diagramme de corps :
Dans le menu, sélectionner la commande *options.push_body* puis cliquer sur l'entité. Le diagramme de corps est chargé. S'il n'existe pas, il est créé (diagramme vide).
Pour remplir le diagramme, insérer le symbole de l'entité, appeler l'interface par la commande *reuse*.
Ajouter les éléments internes et les annoter.
Générer le dictionnaire.

A chaque composant est associé un numéro. Symbole utilisé : *Sequence*. Ce numéro permet de se déplacer plus facilement dans le document généré.

Accès au diagramme de corps :
Dans le menu, sélectionner la commande *options.push_body* puis cliquer sur l'entité. Ou bien appeler le menu associé au champ *Diagram* du *Control Panel* et choisir le diagramme.

4.4.5. Sous–application dégénérée

Comme c'est indiqué plus haut, une sous–application masque son organisation interne à son environnement. Il peut parfois être utile de lever ce principe :

par exemple, il est possible d'introduire une sous–application d'interface entre deux sous–applications; cette sous–application contient par exemple les FIFOs, ... et les renommages (et ne contient que cela);

cette proposition a des avantages certains en terme de réutilisation de l'une ou de l'autre des sous–applications, puisque la communication (qui dépend du contexte) en est exclue; c'est aussi un moyen élégant d'éviter les cycles d'utilisation entre sous–applications;

par contre, elle n'est pas dans l'esprit "CSC = étape d'intégration", car, en général, les "mécanismes" de communication externes ne dépendent pas (ou très faiblement) les uns des autres, donc l'intégration n'a pas de sens réel;

le problème pour ce type de sous–application est qu'il est nécessaire de voir toutes les entités d'interface; sinon, le nombre d'opérations externes à la sous–application serait énorme et serait assez proche de la liste complète des opérations des composants de cette sous–application, ce qui serait illisible et sans intérêt.

Dans ce cas, l'annotation "visibilité" doit être remplie. De plus, la traduction Ada sera très probablement de la première forme citée ci–après.

4.4.6. Traduction en C ANSI

4.4.6.1. Sous–application (stricte)

La sous–application proposent des opérations qui sont tout simplement des opérations définies dans des sous–applications ou des machines ou des objets non stricts contenus dans la sous–application.

Le corps de l'opération n'est autre que le corps d'une opération d'une machine ou d'une opération propre à une sous–application non c.

La sous–application est constituée d'un fichier "spécification" et d'un fichier "corps". Le fichier "spécification" de la sous–application "appli" se nomme "appli.h" et le fichier "corps" se nomme "appli.c".

1. Fichier spécification "appli.h"

   Il décrit les interfaces de ses opérations dont les corps sont ceux d'opérations de machines ou d'objets non stricts.

   De ce fait à chaque opération est associée un pointeur de fonction constant qui pointe sur l'opération ciblée de la machine ou de la sous–application non stricte.

   ```
   extern const void (* appli_remplir) (int);
   ```

2. Fichier corps "applic.c"

   Le fichier corps contient les incudes des machines ou des objets non stricts dont il rend visible les opérations et les déclarations initialisées des pointeurs de fonctions.

   ```
   const void (*appli_remplir) (int) = reacteur_remplir;
   ```

3. Le coût de l'indirection

   L'appel par "appli_remplir" est plus couteux en temps d'exécution que l'appel direct "reacteur_remplir". Le coût en temps d'exécution est évalué à 30% en plus pour la partie changement de contexte. Il est bien évident que l'indirection n'entraine aucun coût supplémentaire en temps d'exécution dans la partie algorithme de l'opération. Dans ce type d'opérations la part du changement de contexte est négligeable par rapport à la part qui revient à l'algorithme, de ce fait ce sur–coût est sans objet.

4.4.6.2. Sous–application non stricte

La sous–application non stricte se comporte comme une machine et comme une sous–application.

Ses opérations sont:

1) soient des opérations propres à la sous–application non stricte et sont déclarées comme les opérations d'une machine;
2) soient des opérations semblables aux opérations d'une sous–application et seront traitées comme celles des sous–applications.

De même les données de la sous–application non stricte seront traitées comme les données des machines.

4.5. Le type abstrait de données

4.5.1. Introduction

Un type abstrait de données est l'entité de description d'une classe d'objets passifs. Il décrit à la fois la nature de l'information portée par toute instance du modèle et les possibilités de comportement d'une telle instance.

Un type abstrait est en général associé à une information "dynamique" (information reçue par l'application, exploitée, modifiée, transmise, voire créée et/ou supprimée).

Un type abstrait est donc une entité nommée

- représentant une "catégorie" d'objets, dont elle définit la structure mais qu'il est inutile de connaître (en général) pour utiliser des instances,
- contenant la définition d'un ensemble d'opérations, définies sur tout objet appartenant à cette classe, qui définissent donc le comportement possible des instances de la classe dont le type abstrait est la description,
- contenant la définition d'un ensemble de constantes de cette classe, c'est–à–dire de valeurs nommées de cette classe (appelés littéraux du type),
- telle qu'il est possible de créer des variables, instances de ce type, manipulables uniquement par les opérations définies dans le type.

Une autre entité peut contenir des instances de ce type (donc déclarer des variables ou des constantes de ce type). Sur ces instances, elle ne pourra utiliser que les opérations définies dans le type abstrait, car elle ne connait pas la structure interne de ces instances. L'avantage de l'entité type abstrait de données est évident : si un jour on décide de changer la représentation interne du type (la nature des informations ou leur structuration), seul le type abstrait associée sera modifié et la modification sera peu, et en général pas du tout, perçue par le reste de l'application. La modification étant correctement localisée et les limites de la propagation de la modification maîtrisées, le test de la modification sera par conséquent très localisé.

Une donnée (c'est à dire une instance d'un type abstrait) est toujours encapsulée.

Pour une donnée, ce qui importe d'abord, au niveau de son environnement, c'est de caractériser ses possibilités de comportement, donc d'utilisation. Sa structure, c'est à dire son organisation interne, est secondaire.

Une donnée peut apparaître comme locale :

- à un (corps de) processus ou de machine : elle est alors une donnée rémanente, c'est à dire qu'elle existe pendant la durée de l'application;
- à une opération, en tant que donnée locale du corps de l'opération ou en tant que paramètre : elle est alors une donnée volatile, c'est à dire qu'elle disparaît à la fin de l'exécution de l'opération.

Durant une conception, le concepteur peut être amené à introduire deux types abstraits de données ayant une forte dépendance l'un envers l'autre. En particulier, certaines opérations peuvent manipuler des instances de chacun des types (autrement dit avoir pour paramètre une instance de chacun des deux types de données).

Il peut donc parfois être utile à un concepteur de regrouper deux ou plusieurs types abstraits en un seul entité.

Ce regroupement de types abstraits sera malgré tout appelé, par abus de langage, un type abstrait.

Aussi une entité "type abstrait" peut être soit la description d'une classe d'objets, soit d'un ensemble de telles classes d'objets. La différence sera de toute façon visible au niveau graphique, puisque plusieurs types de données apparaîtront sur l'interface du type abstrait de données.

Le choix d'un tel regroupement de types abstraits au sein d'une même entité doit être justifié, car cela peut être une gène pour la réutilisabilité et introduire des visibilités non nécessaires.

Un type abstrait de données peut modéliser des données protégées contre l'exclusion mutuelle. Le nom du type abstrait de données doit clairement indiquer que ce sont des données protégées (exemple : TA_protect_nom_de_type_abstrait_de_données). L'annotation "Partagé:" doit être remplie à "oui". Il est fortement conseillé également que le nom des données instances de ce type soit lui aussi explicite sur ce point (par exemple : protect_nom_de_donnée); même utilisation de l'annotation "partagée".

Le type abstrait de données apparaît en principe dans une collection de types abstraits.

4.5.2. Interface

L'interface d'un type abstrait est très semblable à celui d'une machine. Il contient en plus un type(s) de données dans l'interface (ou parfois plusieurs).

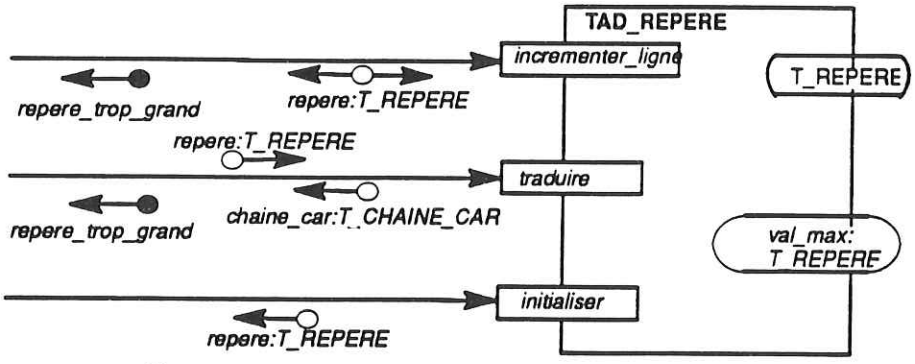


Figure 4.18. exemple d'interface de type abstrait de données

L'interface d'un type abstrait est définie sur un diagramme qui a le même nom.
Utiliser le symbole paquetage.
Le nom d'un type abstrait est préfixé par TAD_.
Tous les composants de ce diagramme sont annotés.
Les paramètres des opérations sont renseignés.
Les annotations d'une opération pourront être reportées si on ouvre un nouveau diagramme pour cette opération.

Annotation obligatoire du diagramme :
Diagram : ta repere
Name : ta repere
Author : dupont
Generated : 11:21:15 1 24 1991

Annotation obligatoire du type abstrait :
paquetage : ta repere
Name : ta repere
Note Description *(éditer un texte de description)*
Requirement : nom1
RequirementName : nom1
RequirementDoc : SRS
Requirement : nom2
RequirementName : nom2
RequirementDoc : Notification de changement 1

Annotation des opérations : *même chose que le type abstrait.*

4.5.3. Environnement

Très semblable à l'environnement d'une machine à la différence suivante importante :

Un type abstrait ne peut utiliser une sous–application, une machine ou un processus.

Cette règle se justifie ainsi : si un type abstrait utilise une machine ou un processus, cela veut dire que toute instance peut utiliser cette machine ou ce processus. Or si ces instances sont situées sur des *Thread of control* différents, il peut donc y avoir partage de (et donc conflit d'accès sur) la machine ou le processus, alors que rien graphiquement ne fait apparaître ce lien : il y a donc risque de problèmes très important. De plus, le fait qu'un modèle puisse utiliser une entité effective n'a pas de sens.

Il est fortement déconseillé de définir des types abstrait de données ayant visibilité sur un type abstrait de processus, car l'expression du parallélisme (et donc le comportement parallèle de l'application) sera très confuse.

L'environnement d'un type abstrait *nom_de_ta* est défini sur un diagramme dont le nom est *nom_de_ta*_env.
Le diagramme seul est annoté.

Création du diagramme d'environnement :
Dans le menu, sélectionner la commande *options.push_env* puis cliquer sur l'entité.
Le diagramme d'environnement est chargé. S'il n'existe pas, il est créé (diagramme vide).
Pour remplir le diagramme, insérer les symboles des entités, appeler les interfaces par la commande *reuse*.
Générer le dictionnaire.

Accès au diagramme d'environnement :
Dans le menu, sélectionner la commande *options.push_env* puis cliquer sur l'entité.
Ou bien appeler le menu associé au champ *Diagram* du *Control Panel* et choisir le diagramme.

4.5.4. Corps

Décrire le corps d'un type abstrait correspond à :

- définir la structure du type de données (au sens codage) : cette description est faite en général par du texte (elle peut être faite graphiquement avec le modèle JACKSON de description des données); pour les structures très complexes, on peut identifier la nature des "composants" du type en conception préliminaire (types abstraits utilisés pour construire le type abstrait), la "syntaxe" étant définie au plus tôt en conception détaillée;
- définir les opérations internes éventuelles (non accessibles par l'environnement), mais utiles pour la réalisation des opérations externes.
- détailler l'algorithmique des opérations (externes et internes).

Un corps de type abstrait ne doit contenir aucune donnée.

La description du corps d'un type abstrait est du domaine de la conception détaillée. Le principe général est que la conception détaillée est décrite dans les fichiers contenant le code source sous la forme d'un PDL, afin de permettre une meilleure cohérence entre le code source et la conception détaillée, en particulier lors de modifications.

Cependant, si le concepteur le juge utile, le corps d'un type abstrait peut être représenté graphiquement, comme le corps d'une machine. Dans ce cas :

Le corps d'un type abstrait *nom_de_ta* est défini sur un diagramme dont le nom est *nom_de_ta*_body.
Seules les données et opérations internes sont annotées.
Les annotations d'une opération pourront être reportées si on ouvre un nouveau diagramme pour cette opération.

Création du diagramme de corps :
Dans le menu, sélectionner la commande *options.push_body* puis cliquer sur l'entité. Le diagramme de corps est chargé. S'il n'existe pas, il est créé (diagramme vide).
Pour remplir le diagramme, insérer le symbole de l'entité, appeler l'interface par la commande *reuse*.
Ajouter les éléments internes et les annoter.
Générer le dictionnaire.

Accès au diagramme de corps :
Dans le menu, sélectionner la commande *options.push_body* puis cliquer sur l'entité. Ou bien appeler le menu associé au champ *Diagram* du *Control Panel* et choisir le diagramme.

4.5.5. Types abstraits dégénérés

Les types abstraits dégénérés sont les suivants :

– entité de déclarations de types et/ou de constantes, sans opérations;
note : il est bon de vérifier si des opérations ne pourraient pas être associées aux types; attention à ces regroupements si les types ou constantes peuvent être vues par des entités différentes qui n'ont pas besoin d'utiliser toutes ces informations, car on induit une mauvaise visibilité qui se paiera lors des modifications;

– entité de déclarations d'opérations : ils sont admissibles dans deux cas uniquement :
— c'est un "complément de type abstrait de données" :
par exemple, il existe un type abstrait par ailleurs, et cette entité est un complément de la liste des opérations associées au type (cela peut être utile si l'utilisation de certaines opérations est réduite à certains "utilisateurs");
un autre exemple est le cas d'un type déjà défini (type prédéfini du langage ou type abstrait apparaissant dans une bibliothèque de composants), pour lequel cette liste d'opération est une liste complémentaire;
— c'est un ensemble d'opérations "à cheval sur deux types abstraits de données", c'est–à–dire qu'on ne peut réellement privilégier l'un des types pour lui associer ces opérations.

4.5.6. Traduction en C ANSI

4.5.6.1. Fichier spécification "MotLong.h"

Dans "MotLong.h" sont décrits les différents types, les opérations proposées aux utilisateurs et les constantes.

a) Les types

```
typedef struct
{       ...
} Mot;
```

La structure est le meilleur moyen pour implémenter les types complexes. Le tableau n'étant pas un type en langage C, les affectations entre tableaux deviennent compliquées.
La déclaration suivante

```
typedef struct
{
 char leMot [...];
} Mot;
```

améliore sans compliquer le code généré les manipulations entre les instances.

b) Les constantes

Les constantes sont préfixées par le nom du type abstrait:

```
extern const mot MotLong_motVide ;
```

```
extern const int      MotLong_tailleMaximum;
```

c) Les opérations

Les opérations sont préfixées par le nom du type abstrait de données et introduites par les prototypes.

```
int    MotLong_taille (Mot const *);
void   MotLong_remplir(Mot *, char*);
```

Les déclarations respecteront les règles suivantes:

1) Pour une meilleure lisibilité l'instance sur laquelle porte l'opération sera toujours passée par pointeur. Une normalisation des appels est des plus souhaitables.
2) Lorsque l'opération ne modifie pas l'état de l'instance, le mot réservé "const" sera utilisé dans la déclaration du paramètre formel associé à cette instance.
3) Le paramètre formel associé à cette instance sera toujours le premier paramètre.

d) Organisation

L'organisation du fichier "spécification" sera la suivante :

Description du module
#ifndef __NOMENTITE #define __NOMENTITE Liste des "include" traduisant la visibilité sur d'autres entités
Les déclarations des types et des constantes typedef struct { char leMot [...];} Mot; extern const mot MotLong_motVide; extern int MotLong_tailleMaximum;
Les prototypes des opérations visibles void MotLong_remplir(Mot *, char*); int MotLong_taille (Mot const *); #endif

4.5.6.2. Fichier corps "MotLong.c"

Les corps des opérations sont déclarés dans le fichier "MotLong.c". Les constantes sont également déclarées dans ce fichier. Toutes les déclarations de données extern repérées dans le fichier "spécification" doivent se retrouver dans le fichier "corps"

Les corps de toutes les opérations déclarées dans le fichier "spécification" doivent figurer dans le fichier "corps". Les opérations supplémentaires sont locales au corps, elles ne seront visibles que par ces opérations supplémentaires et les opérations déclarées en prototype dans le fichier "spécification". Elles sont classées "static".

a) Organisation

L'organisation du fichier "corps" sera la suivante :

#include "MotLong.h"
Description du module
Liste des "include" traduisant la visibilité sur d'autres entités
Les prototypes des opérations locales static int calculerTaille (Mot *);
Les déclarations des constantes const mot MotLong_motVide = ...; const int MotLong_tailleMaximum = ...;
Les types et constantes locales static const int nbEspaces = ...;
Les corps des opérations visibles void MotLong_remplir(Mot *lemot, char* chaine) { ... }
Les corps des opérations locales static int calculerTaille (Mot * lemot) { ... }

Conseil : éviter de définir des opérations sur les types définis dans le Type abstrait en dehors du fichier corps correspondant au Type abstrait.

Il n'est pas possible de voir des déclarations de données dans le fichier "corps". Seules des constantes locales sont possibles. Elles sont classées "static".

b) Fichier "corps" trop gros

L'ensemble des opérations d'un type abstrait peut dépasser le nombre de lignes admis par les organismes de qualité. L'éclatement du "corps" est la seule solution. Il faut que les corps de certaines opérations soient reportés dans d'autres fichiers.

Deux possibilités sont envisageables.

La première consiste à créer un fichier contenant le corps de l'opération déportée et le remplacer dans le fichier corps par un ordre d'inclusion:

```
# include   "MotLong_remplir.i"
# include   "MotLong_taille.i"
```

Dans cette option toutes les opérations peuvent utiliser les constantes locales, les types locaux, et les opérations locales.

La deuxième consiste à créer un fichier contenant le corps de l'opération déportée.

La visibilité sur la spécification sera réalisée par

```
# include   "MotLong.h"
```

Dans ce cas, l'opération n'aura plus de visibilité sur les opérations locales, les constantes locales et les types locaux (qui ne présenteraient plus beaucoup d'intérêt). Le fichier s'appelerait "MotLong_remplir.c".
La combinaison des deux possibilités est bien sur possible.

4.6. Type abstrait de processus

4.6.1. Introduction

Un type abstrait de processus est l'entité de description d'une classe d'objets actifs. Il décrit un modèle de *Thread of control*, c'est–à–dire d'objets de base ayant un comportement parallèle.

Ce genre de modèle est utile lorsque plusieurs processus identiques peuvent fonctionner simultanément dans l'application. Plutôt que de définir le comportement de chacun de ces objets, il vaut mieux introduire ce modèle commun.

Chacune des instances de ce type représente un *Thread of control*, et est donc critique vis à vis du parallélisme. Les instances de ce type doivent donc être clairement identifiées[7]. Il est quasi–obligatoire de les faire apparaître dès la conception préliminaire, afin d'identifier la complexité du parallélisme de l'application et donc du comportement d'ensemble du logiciel. Les instances multiples (tableaux de processus, ...) doivent avoir un nom suffisament évocateur pour bien indiquer qu'elles représentent des instances multiples de processus.

Une approche conseillée est de faire apparaître ces instances dans des entités dédiées (une telle entité peut en contenir plusieurs) à ces instances (un peu comme des instances de génériques), afin de ne pas les mettre dans des machines (car elles ont un comportement passif) ni dans des processus (car le processus est séquentiel ou "mono–*Thread of control*").

Attention, si dans le temps une suite de processus identiques sont créés et supprimés dynamiquement mais qu'il n'y ait qu'un seul processus à un instant donné, il vaut mieux le faire apparaître comme un processus "normal" et non comme une instance d'un type processus, avec des opérations de lancement/création et d'arrêt/suppression, car on montre bien qu'à tout moment, il n'y en a qu'un seul au maximum.

7. Il est possible d'obtenir la quasi–équivalence en remplaçant un type abstrait de processus par un processus générique sans paramètres générique, ce qui présente l'avantage de faire facilement apparaître les instances et de manière claire, et par l'outil. Sa traduction en Ada peut être un type tache, ce qui est quasi obligatoire si ces instances sont crées/supprimées dynamiquement (mais qui n'est pas critique dans le cas contraire).
De plus, si l'instance doit appeler des opérations de machines ou d'autres processus, ces opérations peuvent être passées en paramètres génériques.
La notion de type abstrait de processus peut alors être supprimée.

4.6.2. Interface

Il est conseillé de faire apparaître des opérations de création et de suppression sur l'interface du paquetage définissant le type de processus (même si elle ne seront pas traduites en procédures/fonctions en codage) afin de faire apparaître quelles sont les entités qui les créent et les suppriment et donc avoir un aperçu sur la localisation des instances :

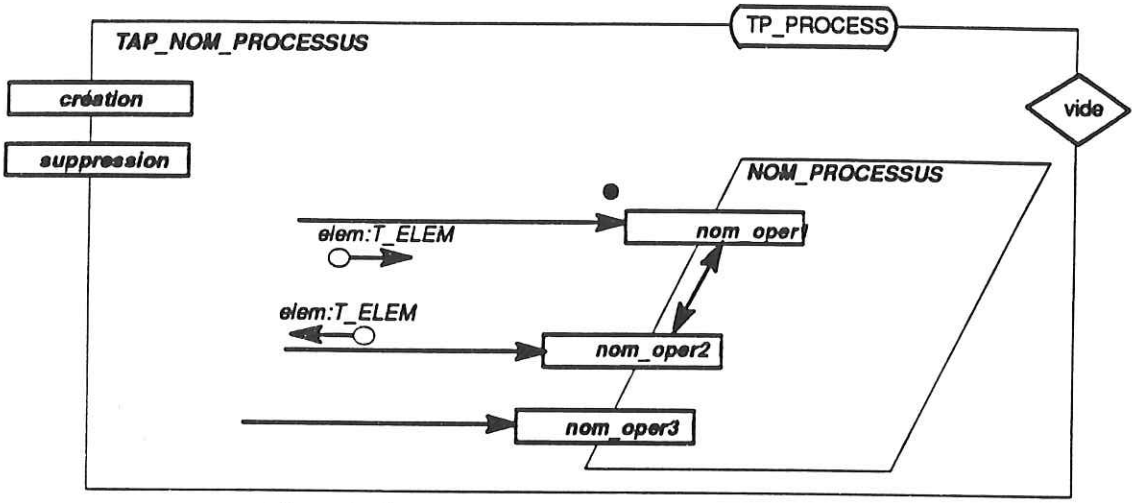


Figure 4.19. Forme de l'interface d'un type abstrait de processus

S'il existe des contraintes sur le nombre d'instances possibles, cette indication peut être portée dans une annotation.

4.6.3. Environnement

Identique à l'environnement d'un type abstrait de données, sans la restriction sur la visibilité sur d'autres types de processus, bien qu'il faille être encore très prudent sur ce point.

4.6.4. Corps

Identique au corps de processus.

L'implémentation est à définir en fonction de l'environnement d'exécution disponible .

4.7. Collection

4.7.1. Introduction

Le fait d'introduire la notion de collection permet de bien indiquer que les descriptions contenues dans une telle collection ne se situent pas au même niveau conceptuel que ce qui est décrit dans les autres entités que sont les sous–applications, objets non strict, processus ou machines.

Dans la description d'une application, on cherche à identifier en premier lieu le parallélisme (par l'intermédiaire des processus) ainsi que les ressources utilisées et leur localisation (réalisée par l'intermédiaire des machines).

A coté de cette "mise en place" de l'application, on décrit la classe, "modélisation" des objets dont plusieurs instances apparaissent dans l'application : on décrit alors ces classes sous forme de types abstraits ou de génériques.

Conceptuellement, les modèles (génériques ou non) "n'existent" pas dans l'application : ce qui "existe", c'est l'ensemble des instances (variables ou constantes) qui se manipulent selon la description donnée dans le modèle.

Les collections sont éventuellement décomposables en sous–collections si c'est nécessaire ou en types abstraits. Cela permet en particulier des regroupements logiques de modèles.

Conseil : comme pour la sous–application et la sous–application non stricte, il faut éviter de décomposer une même collection de manière non homogène en collections et en types abstraits (cf 4.4.1.).

4.7.2. Interface

L'interface d'une collection contient la liste des composants de la collection pour donner une première visibilité du contenu de la collection. Il est clair que le principe, au niveau d'une collection, n'est surtout pas le masquage total, d'où cette présence de l'identité des composants. Par contre, la description totale de l'interface des composants est reportée au corps de la collection.

La collection sera représentée par un paquetage avec un préfixe particulier : CO_. Il est possible de différencier les collections de génériques des autres collections en introduisant les préfixes CO_TA_ pour les collections de types abstraits et CO_GEN_ pour les collections d'entités génériques.

Ces préfixes sont bien entendus réservés. Il est interdit de s'en servir pour d'autres utilisations.

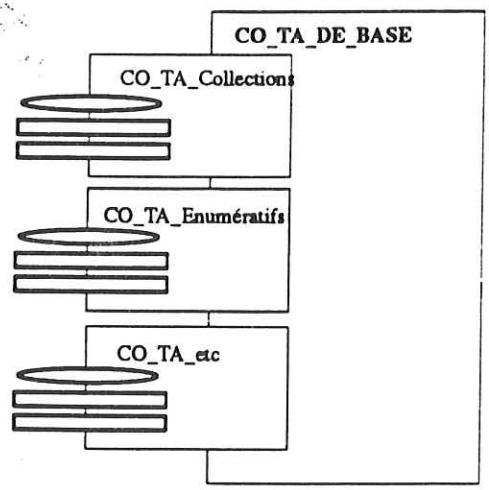


Figure 4.20. Exemple d'interface d'une collection se décomposant en collections

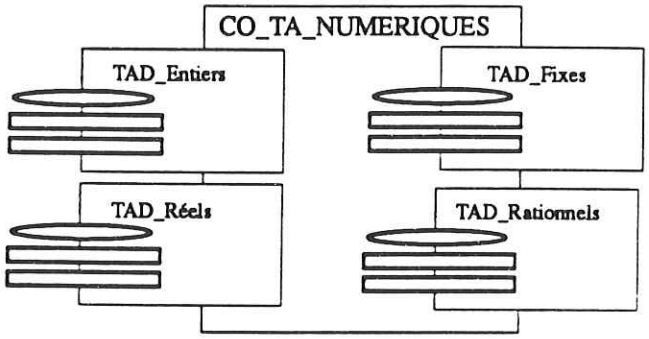


Figure 4.21. Exemple d'interface d'une collection se décomposant en type abstraits

Annotation des opérations : *même chose que le type abstrait.*

L'interface d'une collection est définie sur un diagramme qui a le même nom.
Utiliser le symbole paquetage.
Le nom de la collection est préfixé par CO_.
Seul l'entité collection est annotée.

Annotation obligatoire d'une collection :
paquetage : CO_non_collection
Name : CO_non_collection
Note Description *(éditer un texte de description)*

4.7.3. Environnement

Il est réduit à des liens vers d'autres collections, sur lesquelles on fait apparaitre uniquement les composants utilisés.

4.7.4. Corps

Le corps d'une collection contient les interfaces des descriptions des modèles d'objets (types abstraits et génériques) ou des (sous–)collections contenus dans la collection. Le terme *corps* est abusif puisqu'il présente en fait l'interface complète, ainsi que les liens entre les composants.

Corps d'une collection composée de sous–collections :

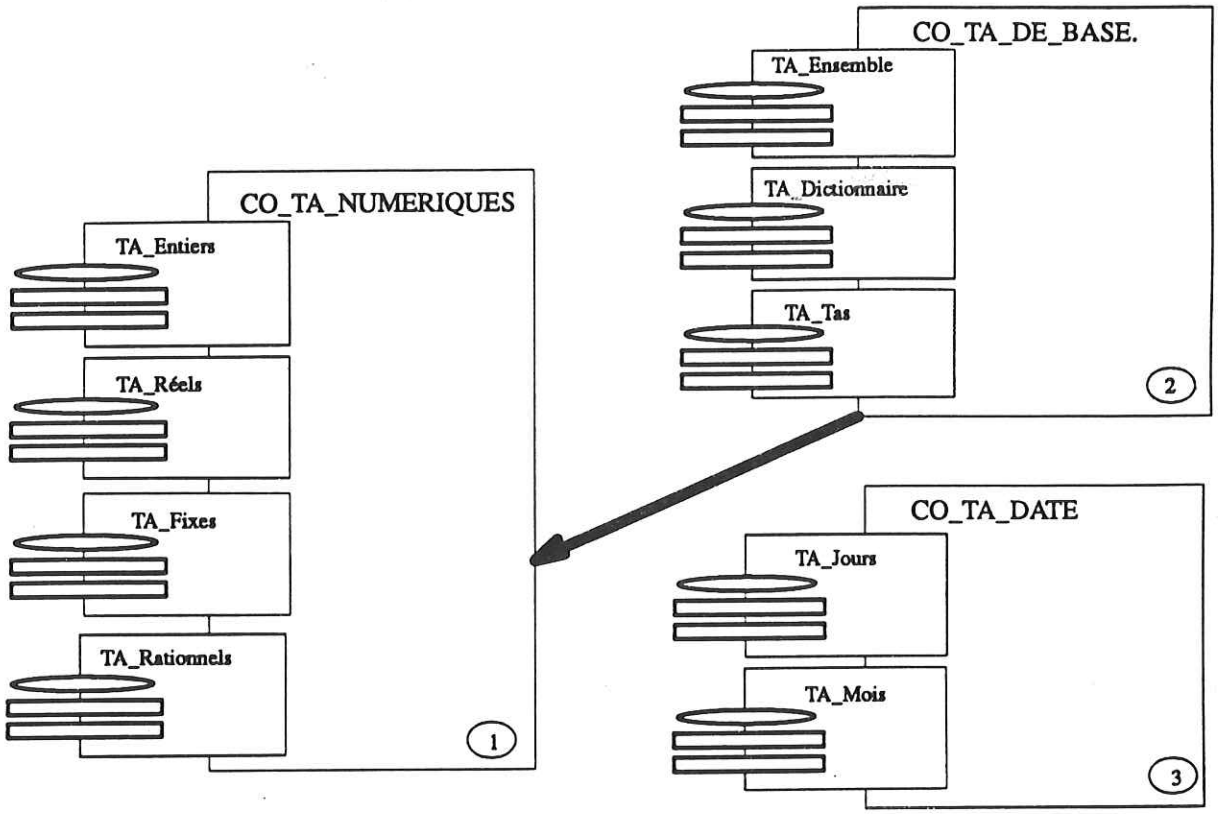


Figure 4.22. Corps de collection composé de collections

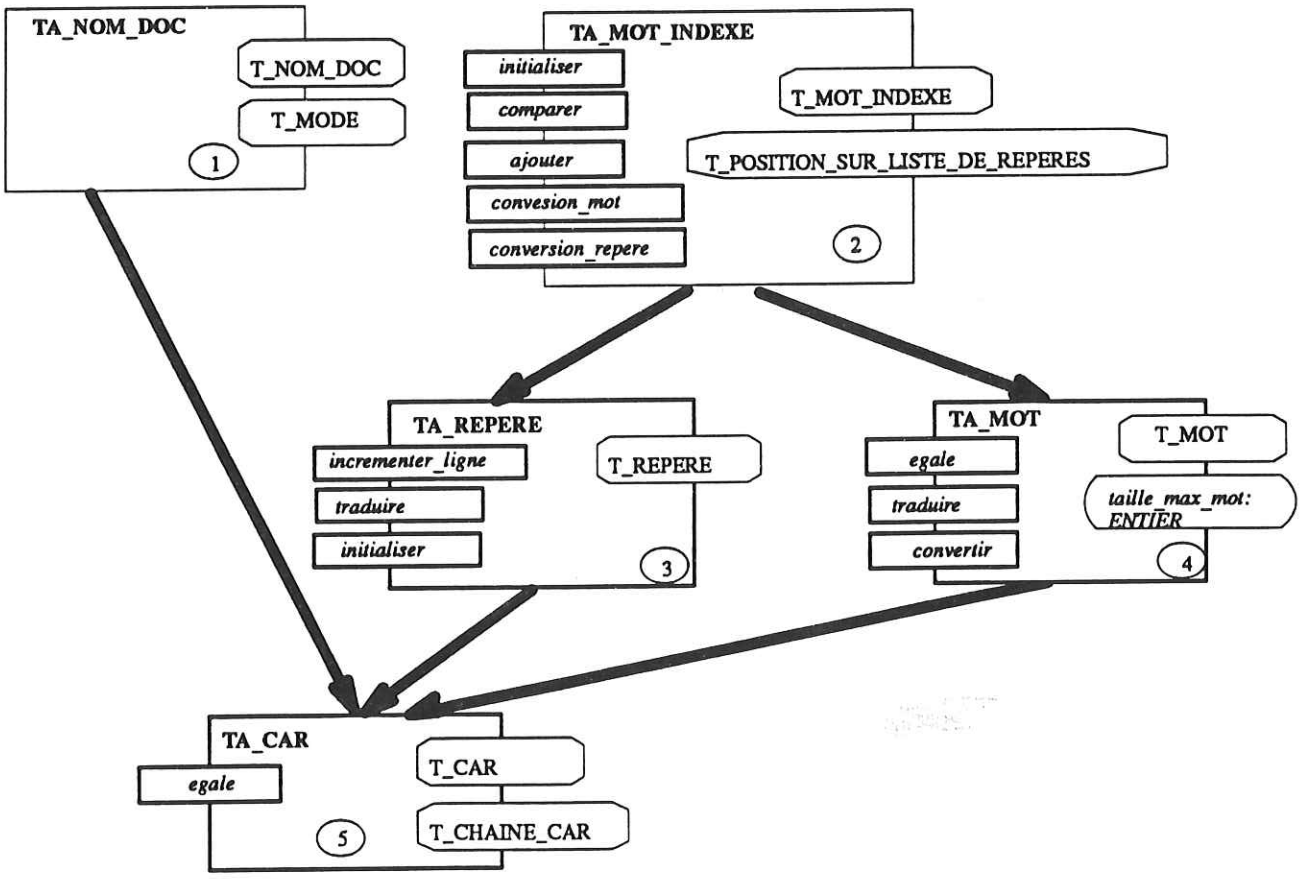


Figure 4.23. Corps d'une collection composé de types abstraits

Le corps d'une collection *nom_de_collection* est défini sur un diagramme dont le nom est *nom_de_collection*_body.
Ses composants ne sont pas annotés puisqu'ils sont définis dans d'autres diagrammes.

Création du diagramme de corps :
Dans le menu, sélectionner la commande *options.push_body* puis cliquer sur l'entité. Le diagramme de corps est chargé. S'il n'existe pas, il est créé (diagramme vide).

Pour remplir le diagramme, insérer les symboles des constituants, appeler leur interface par la commande *reuse*.
Générer le dictionnaire.

A chaque composant est associé un numéro. Symbole utilisé : *Sequence*. Ce numéro permet de se déplacer plus facilement dans le document généré.

Accès au diagramme de corps :
Dans le menu, sélectionner la commande *options.push_body* puis cliquer sur l'entité. Ou bien appeler le menu associé au champ *Diagram* du *Control Panel* et choisir le diagramme.

4.7.5. Traduction C ANSI

La collection est un pur élément de conception et n'est donc pas traduit en C. Le code correspondant à une collection est un ensemble de fichiers correpondant à une mise à plat des traductions en C des entités contenues dans la collection.

5. LIENS AVEC LE SDD

5.1. Phasage : conception préliminaire/conception détaillée

En conception préliminaire, il s'agit d'identifier et de passer en revue les choix de conception liés aux CSCs.

En fait, trois approches sur ce qu'il est nécessaire de fournir dans le SDD en conception préliminaire sont possibles qui dépendent de la taille du CSCI à développer :

- l'approche de référence est la suivante :
  * remplir le chapitre 3 de conception préliminaire (sur les CSCs, donc les entités composites);
  * fournir les interfaces des CSUs (entités de base) pour améliorer la compréhension (et donc faciliter l'évaluation) seulement, sans valeur d'engagement ni de complétude;

    il est clair qu'afin "d'assurer" la conception préliminaire, le concepteur est souvent "en avance de phase" et a déjà commencé sa reflexion sur la conception détaillée avant la revue de conception préliminaire;
- l'approche pour les logiciels importants (taille à déterminer par l'unité) :
  * remplir le chapitre 3 de conception préliminaire (sur les CSCs, donc les entités composites) uniquement;
- l'approche pour les logiciels de petite taille (taille à déterminer par l'unité) :
  * remplir le chapitre 3 de conception préliminaire (sur les CSCs, donc les entités composites);
  * fournir les interfaces des CSUs (entités de base) avec valeur d'engagement et de complétude.

5.2. Schémas de OOSD dans le SDD

On appelera par la suite CSC *terminal* un CSC (correspondant à une sous–application, un objet non strict ou une collection) ne se décomposant qu'en CSUs (correspondant à des entités de base).

Dans le chapitre 3 du SDD, conception préliminaire, les schémas d'interface, de corps et d'environnement (s'il est nécessaire) d'un élément correspondant à un CSC sont inclus dans le SDD, dans la description du CSC correpondant, sauf pour les CSCs terminaux pour lesquels le diagramme de corps n'apparaît pas dans le chapitre 3 du SDD.

Dans le chapitre 4 du SDD, conception détaillée, le diagramme de corps des CSCs terminaux apparaît dans le paragraphe correspondant au CSC. Les diagrammes d'interface, et si nécessaire de corps et d'interface d'un élément correspondant à un CSU, apparaissent dans le sous–paragraphe consacré au CSU.

Note :

le principe de base pour la conception détaillée, est de générer le maximum d'informations pour le SDD à partir des fichiers de code source (par extraction de commentaires – à définir plus précisemment les besoins en informations), afin de faciliter la mise à jour de la conception détaillée lors du codage et du test.

A priori, les informations du SDD concernant le corps des opérations, voire des objets de base, sera donc extrait des fichiers du code source.

5.3. Génération du SDD

Il s'agit de définir le principe de la répartition de ce qui doit être réalisé sous TPS, ADE et les fichiers de code source pour la génération documentaire.

1. Domaine d'application — TPS

2. Documents cités en référence — TPS

3. Conception préliminaire

3.1 Généralités

le lien avec les autres CSCIs/HWCIs peut éventuellement être basé sur le diagramme de contexte de SA–RT. Il est également envisageable de proposer un diagramme de contexte sous OOSD, en utilisant des SAs pour représenter les CSCIs/HWCIs.

3.1.1 Architecture du CSCI

3.1.1.1 Organisation statique

citer tous les CSCs plus un diagramme des NDSs du CSCI — ADE

3.1.1.2 Architecture dynamique

à définir – en l'état actuel : — TPS

3.1.1.3 Interfaces entre CSCs

diagramme de premier niveau — ADE

3.1.2 Etats et modes du système

à partir des annotations etats/modes systèmes + texte — TPS + ADE

3.1.3 Allocation mémoire ...

à partir des annotations memoire ... + texte — TPS + ADE

3.2 CSC_X1 — ADE

.....

4. Conception détaillée

4.X CSC_X — ADE

4.X.Y CSU_Y

diagrammes — ADE

algorithmique + divers — code

5. Données du CSCI

il s'agit des données globales : normalement en COO, il ne doit pas y en avoir.
il peut être judicieux d'introduire ici les données de paramétrages (données d'installation, de site ...) et de citer leur localisation dans les CSCs/CSUs.
Ces données pourraient être extraites de OOSD, en utilisant une annotation spéciale et extraites du code ADE/code

6. Fichiers du CSCI code

7. Traçabilité ADE

8. Notes TPS

6. DEMARCHE

6.1. Introduction

Dans l'approche orientée par les objets, on se rapproche de notre approche naturelle pour résoudre un problème autre qu'informatique.

Par exemple, si quelqu'un veut résoudre un problème avec sa voiture (amélioration, dépannage ...), il perçoit d'abord le problème comme étant un problème fonctionnel : ma voiture doit faire telle chose. A partir de là, il essaie d'identifier les composants de sa voiture qui peuvent intervenir de près ou de loin dans cette fonction (carburateur, ...). Ensuite, il évalue en quoi ces composants participent à la fonction. Il définit les liens entre ces composants et imagine la solution. Il réalise cette solution en modifiant ou en ajoutant des composants.

Dans l'approche orientée par les objets pour le développement de logiciels, la démarche est la même, à ceci près que à priori tout est à créer (sauf en cas de réutilisation) :

Entrée : le logiciel doit répondre à une capacité (fonctionnalité + performances associées) donnée.

– le concepteur identifie les objets intervenant dans cette capacité;

– puis il caractérise ces objets : structure (informations) et comportement (dans un premier temps les fonctionnalités prises en charge, dans un deuxième temps les opérations permettant ces fonctionnalités et donc la vision externe du comportement "déclenchable" de l'environnment);

– il définit ensuite les liens entre ces objets;

– il complète son "modéle" (en ajoutant éventuellement de nouveaux objets et/ou leurs opérations associées).

6.2. Quelques remarques introductives

6.2.1. ... A propos du séquencement des activités :

Le séquencement des activités de la démarche est un guide des activités de conception. Il serait utopique d'en faire un absolu (donc des étapes). Le concepteur aura parfois besoin de réaliser certains objets, ne serait–ce que pour s'assurer de la faisabilité de sa solution ou parce que la structure de ces objets lui est imposée par la spécification et qu'il n'est alors pas utile de retarder la réalisation.

De plus, une conception est faite d'avancées et de reculs (correction, remises en cause au fur et à mesure de l'avancement). Ce qui est présenté est donc plus l'esprit de la démarche qu'un enchaînement à suivre strictement.

De plus, les questions proposées au concepteur pour l'aider dans sa démarche ne prétendent pas être exhaustives, d'autant que les questions "réelles" dépendent très souvent de l'application en cours de conception.

6.2.2. ... A propos de la transition spécification/conception, ... ou désorganiser pour réorganiser :

Dans le cadre du cycle de développement du logiciel, la spécification décrit des besoins, exprimés par le "client" (ou au moins sur lesquels il donne son avis), sur ce que doit faire le logiciel. Lorsqu'il y a accord entre le client et le développeur sur cette expression, ces besoins deviennent des exigences de développement.

En théorie, cette spécification ne doit pas contenir de choix sur le "comment" le logiciel répondra à ces exigences. Elle adopte donc le "point de vue client/utilisateur".

Il n'y a donc aucune raison pour que l'*organisation* de l'expression des exigences (définition du problème), dans la spécification, corresponde de près ou de loin à l'*organisation* du logiciel définie dans la conception (solution du problème). Par contre, les exigences peuvent devenir des critères pour les choix de conception (en particulier les exigences de facteurs qualité).

6.2.3. ... A propos de la justification des choix :

Un principe général à toute la démarche (et qui ne sera pas rappelé par la suite) est qu'

Il faut toujours justifier les choix de conception !

Ces justifications des choix doivent être réalisées au fur et à mesure que le concepteur effectue ces choix (cela suppose un outil documentaire "en ligne").

Une justification peut concerner par exemple le/les critères qui ont été utilisés pour regrouper des objets dans un objet composite. Les justifications des choix peuvent être rapportées dans l'annotation JUSTIFICATION.

6.2.4. Quand peut–on réellement commencer l'organisation du logiciel ?

La conception du logiciel correspond à deux objectifs :

– trouver une solution (logique) au problème,

– organiser cette solution.

La réponse à la question du titre ci–dessus dépend donc de la question suivante : la solution logique est–elle exprimable directement à partir de la spécification ou faut–il introduire un *"modèle" de raisonnement* (ou *modèle de calcul*)?

En effet, si le calcul réalisé par le logiciel nécessite d'introduire un modèle mathématique de calcul, un modèle de thermodynamique, ... et si la spécification n'est pas exprimée en termes de ce modèle de calcul, il est nécessaire de définir d'abord ce modèle, de l'appliquer au problème à résoudre, afin d'avoir une définition du problème à résoudre en termes de ce modèle de calcul.

L'organisation du logiciel ne pourra commencer réellement qu'à partir de ce moment. Il est illusoire d'organiser le logiciel si le concepteur n'a aucune idée du fait que tout le calcul va s'appuyer sur des équations de calcul différentiel, de produits matriciels, ...

C'est le même problème qui va se poser si le logiciel est de nature gestion et que le graphe de dépendances (ou de relations) entre les entités manipulées n'est pas (ou peu) défini, parce que l'expression fonctionnelle a été considérée comme suffisante par le spécifieur et/ou le client. Dans ce cas, il est nécessaire d'introduire un modèle de dépendances entre les entités pour en mesurer toute la complexité (en utilisant le modèle entité/association par exemple).

Dans bon nombre d'applications, la spécification est déjà exprimée en termes de ce modèle de calcul. Dans ce cas, la solution logique pourra être définie simultanément avec l'organisation du logiciel, et la découpe du logiciel liée à cette organisation sera un bon moyen de découper la définition de la solution du problème.

Dans le cas contraire, il faut d'abord faire une première abstraction du problème concret qui soit sa traduction dans un modèle mathématique, physique, relationnel ... Cette étape d'abstraction du problème dans un modèle de calcul reste du domaine de l'expression du problème, mais dans un modèle à partir duquel le concepteur pourra raisonner, donc trouver la solution et sur lequel s'appuiera le traitement réalisé par le logiciel.

Cette étape d'abstraction peut être considérée comme faisant partie de la spécification, puisqu'il s'agit bien toujours de l'expression du problème. Mais si elle n'a pas été réalisée durant la phase de spécification des exigences, le concepteur devra la réaliser car c'est l'étape préalable à la découverte d'une solution logique.

6.3. La démarche de conception

6.3.1. Niveau système

L'organisation du logiciel débute dès la conception système lors de l'organisation en CSCIs.

On ne donnera pas ici les critères de sélection des CSCIs qui sont basés essentiellement sur des critères techniques (identification des performances, caractérisation en termes de capacité vue par le client, complexité des interfaces, unité de qualification ...), de gestion des configurations (spécification dédiée, niveau de gestion des modifications, niveau de suivi des états, gestion des interfaces ...) et de gestion de projet (unité de gestion donc de coût, de délai et de suivi du développement, niveau de détail de visibilité par le client sur le déroulement du développement, ...).

Le schéma ci–après donne une vision synthétique des activités et des critères de décision liés à ces activités qui visent à optimiser le coût du développement du logiciel.

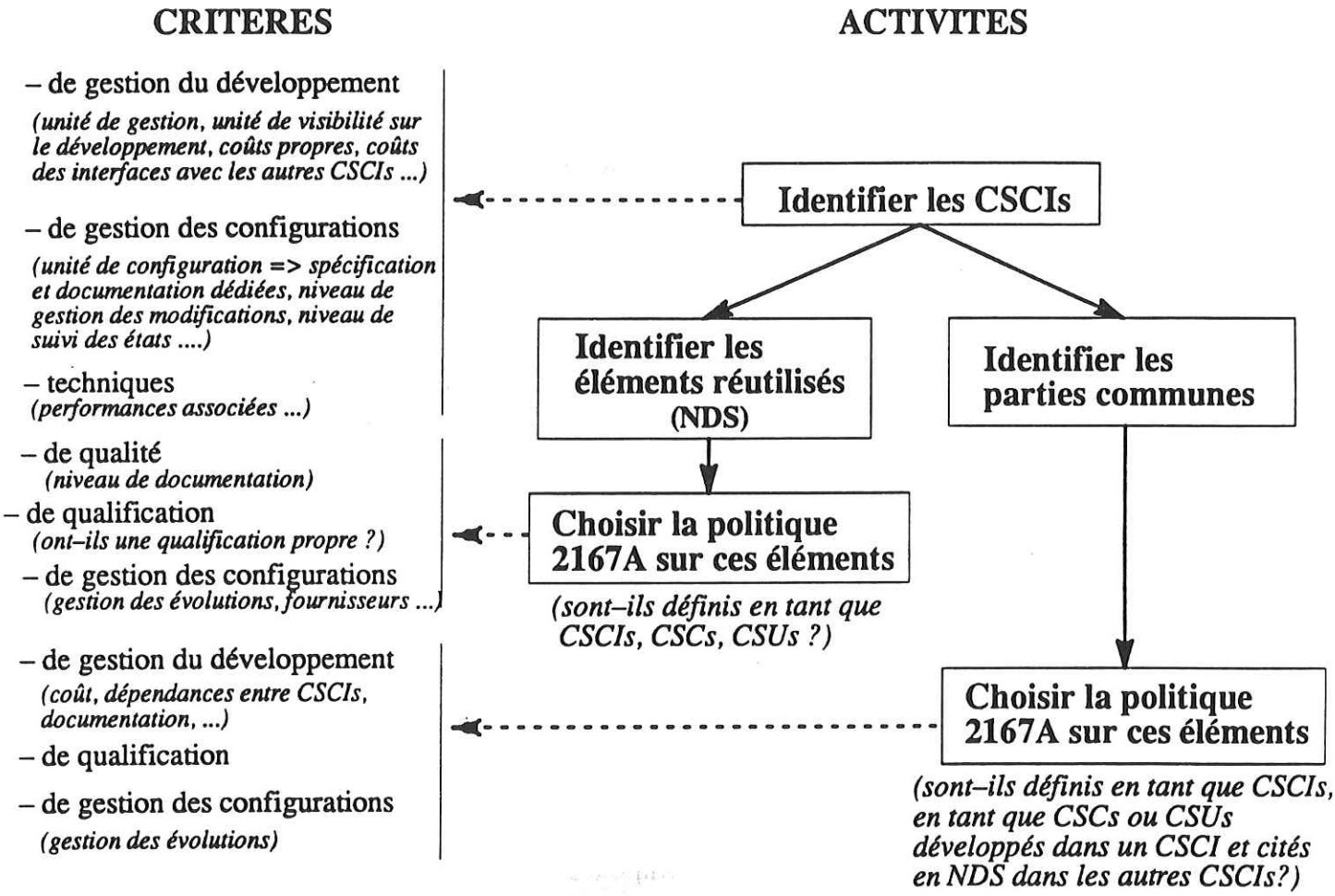


Figure 6.1. Schéma général de gestion de la réutilisation au niveau système

La découpe du logiciel en CSCIs peut être faite en décomposant une sous–application représentant le logiciel en sous–applications et/ou des collections identifiés comme des CSCIs.

6.3.2. Niveau CSCI

La conception du CSCI va se réaliser par niveaux de détail :

- le niveau le plus haut (premier niveau de décomposition du CSCI) va être constitué d'entités implémentant des "macro–objets";
- à chaque entité implémentant un macro–objet est associée une décomposition en entités plus "détaillées" (qui peuvent implémenter encore des macro–objets);
- ce procédé de décomposition se poursuit jusquà obtenir des entités de base.

Le nombre d' entités introduites à chaque décomposition doit être limité (3 à 9 maximum, à définir pour le projet).

Les objets découverts à un niveau de détail ne sont pas forcément les "macros–objets" attendus (en général, ils sont plus nombreux ... et souvent plus petits que ceux désirés), aussi le concepteur doit–il faire un travail d'organisation pour répartir tous les objets découverts en des "macro–objets". Les objets découverts à un niveau de détail peuvent nécessiter plusieurs niveaux d'organisation en macro–objets, afin d'avoir une complexité en nombre d'objets qui soit toujours raisonnable. C'est ce qui est présenté par le schéma suivant :

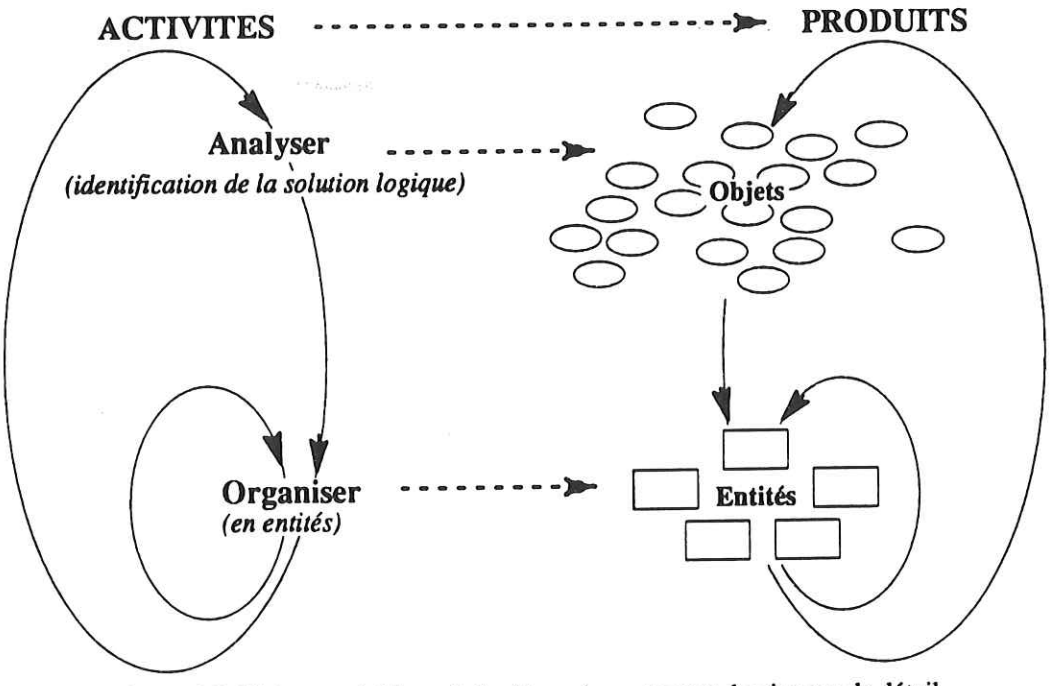


Figure 6.2. Vision synthétique de la démarche en termes de niveaux de détail

Le concepteur doit travailler par niveaux de détail homogènes, c'est–à–dire ne pas développer une architecture de conception en profondeur jusqu'au détail pour certains entités alors que d'autres ne sont pas encore développés (afin de minimiser les modifications qui apparaissent inévitablement lorsqu'on décompose).

Ces deux "étapes" dans la décomposition d'un logiciel (analyse et organisation), ainsi que leurs produits (objets plus ou moins intuitifs et entités organisés, respectivement) sont souvent dites de l'*espace du problème* et de l'*espace de la solution*, respectivement.

Selon la complexité du problème, l'analyse peut être plus ou moins élaborée, et nécessiter une expression plus ou moins formelle. Par exemple, une fiche de description d'objet peut être utilisée comme celle donnée dans la méthode MAC_ADAM–STROOD. Attention toutefois de savoir intégrer ces informations sous ADE.

Par contre, avant d'organiser les objets en entités, il est souvent nécessaire d'avoir un "brouillon" de la visiblité entre les objets, afin de faire une bonne répartition. Parfois le concepteur aura besoin de "dessiner" cette visibilité avant d'organiser, parfois sa compétence du domaine (et son intuition) lui permet de prendre les décisions sans avoir une vision trop précise de ces liens. C'est vrai en particulier dans le (les) premier(s) niveau(x) de décomposition, dans le(s)quel(s) le nombre d'objets ne permet pas d'avoir une vision très précise, et pour le(s)quel(s) le concepteur doit organiser "a priori" afin de maîtriser la complexité, quite à revenir sur ces choix s'il se rend compte d'une mauvaise organisation en entités.

La figure 6.3. suivante donne les grandes lignes de la démarche de conception d'un logiciel.

Il peut paraître surprenant que l'analyse approfondie du parallélisme intervienne assez tard. Deux remarques à ce sujet :

- pour justifier ce choix, nous allons faire une comparaison avec une gestion de projet – qui ressemble beaucoup à une conception de logiciel, les individus correspondant aux processus (actifs) et les tâches aux fonctionnalités à remplir –;

  il serait aberrant qu'un chef de projet définisse le nombre de personnes dont il a besoin dans son projet avant d'avoir une vision raisonnablement fine de l'ensemble des travaux à réaliser; un chef de projet définit donc d'abord son projet, puis il identifie les ressources humaines nécessaires pour le mener à bien; pour le chef de projet l'identification des ressources humaines est très importante car c'est ce qui détermine le coût (et les délais) de réalisation de son projet, donc il attend d'avoir des données suffisantes pour prendre de bonnes décisions;

  ce qui est proposé en conception est de même nature : le concepteur doit avoir une vision suffisamment fine des activités à effectuer (activités traduites par des opérations des objets) avant de déterminer le parallélisme nécessaire;

  "le parallélisme est très important dans un logiciel temps réel" : partant de ce constat (parfaitement justifié), la plupart des méthodes proposent d'identifier dès le début de la conception le parallélisme (donc les processus et leurs synchronisations) nécessaire : encore une fois, si l'on fait le parallèle avec la gestion de projet, c'est risqué; cela conduit en général à un parallélisme trop complexe (trop de processus) donc des risques en performance, en test de mise au point et en évolution;

  dans l'approche proposée ici, "le parallélisme est très important", donc le concepteur doit attendre d'avoir des données suffisantes pour décider du parallélisme nécessaire;

par contre, il est clair que lorsque le parallélisme est fixé, cela a un impact sur les interfaces des entités déjà introduits; il ne faut donc pas attendre le dernier moment pour le définir (le niveau de détail extrême n'est en général pas utile), même si la figure 6.3. semble le montrer; en particulier, dans la majeure partie des cas, il ne faut pas attendre la description détaillée des machines ou des types abstraits; malheureusement, il n'est pas possible de fixer des critères précis sur le niveau de détail idéal pour se poser les problèmes de parallélisme : ce choix est donc laissé au concepteur;

en fait, il existe un parallélisme que le concepteur peut identifier très tôt, comme par exemple le parallélisme lié aux entrées/sorties, ... et un parallélisme qu'il a intéret à définir plus tard : celui qui est intrinsèque à l'application; notons également que pour identifier la maîtrise du transfert (cf figure 6.3.), il est souvent nécessaire d'avoir une idée sur l'aspect actif/passif des entités;

- lors de la décomposition de l'application, le concepteur peut s'appuyer sur un parallélisme "intuitif", en particulier lors de l'application de scenarios de fonctionnement, mais sans chercher à fixer les aspects synchronisation.

CRITERES TECHNIQUES

– à partir de la spécification
(analyse de l'énoncé basé sur le vocabulaire : entités du monde réel modélisées, informations échangées entre fonctions ...
Pour SA–RT : identification des terminaisons, des stockages, des flots de données ...)

– informations de conception
(même approche mais basée sur une analyse du "comment réaliser" les fonctions spécifiées et attribuées aux objets)

Objectifs :
. obtenir une forte cohésion et un faible couplage dans et entre les regroupements crées,
. optimiser le coût en test ,
. optimiser les coûts des évolutions.

Points de vue liés:

– objet
(regroupement logiciel image d'un regroupement existant dans le monde réel)

– fonctionnel
(regroupement d'objets participant à une même fonctionnalité –> évolutivité, coûts des tests de non–régression)

– lien avec l'intégration
(regroupement étape d'intégration logicielle ?, étape d'intégration sur le système ?)

– lien avec l'architecture matérielle
(répartition multi–UC, besoins et donc performances en communication ...)

– interfaçage
(regroupement de la couche d'interface. souvent lié à des contraintes sur l'interface
** attention au couplage/cohésion, tests fonctionnels, intégration logicielle)*

– criticité
(éviter de regrouper des éléments de criticités différentes : réutilisabilité, sécurité, sureté de fonctionnement, performance, degré de stabilité des spécifications, des interfaces ...)

ACTIVITES

Fixer l'interface de l'application

Rédiger l'IDD

Identifier les candidats objets

qui pourront devenir des objets –donc ayant un état– ou des parties d'objets (c'est–à–dire non décrites individuellement) –donc entrant dans la définition de l'état d'un objet–

Les répartir en entités "nuages"

répartir les informations d'état en objets composites
définir les (principales) fonctionnalités prises en charge par les entités

Repérer les entités communes

objets partagés (doivent–ils être extraits ou non ?) –> machines, sous–applications partagés
données ou modèles communs –> types abstraits et collections
généricité

Repérer les entités réutilisées

recherches, analyses comparatives de coût

Repérer les entités réutilisables

surcroit d'analyse
surcroit de documentation
traduits en types abstraits ou en génériques

Eléments de GESTION

– gestion du développement :
s'interfacer avec les équipes de réalisation des autres CIs

– gestion des configurations :
maîtriser les évolutions des interfaces (niveau système)

– gestion du développement :
répartir (éventuellement) l'analyse de conception entre les membres de l'équipe

– gestion du développement :
prévoir réunions

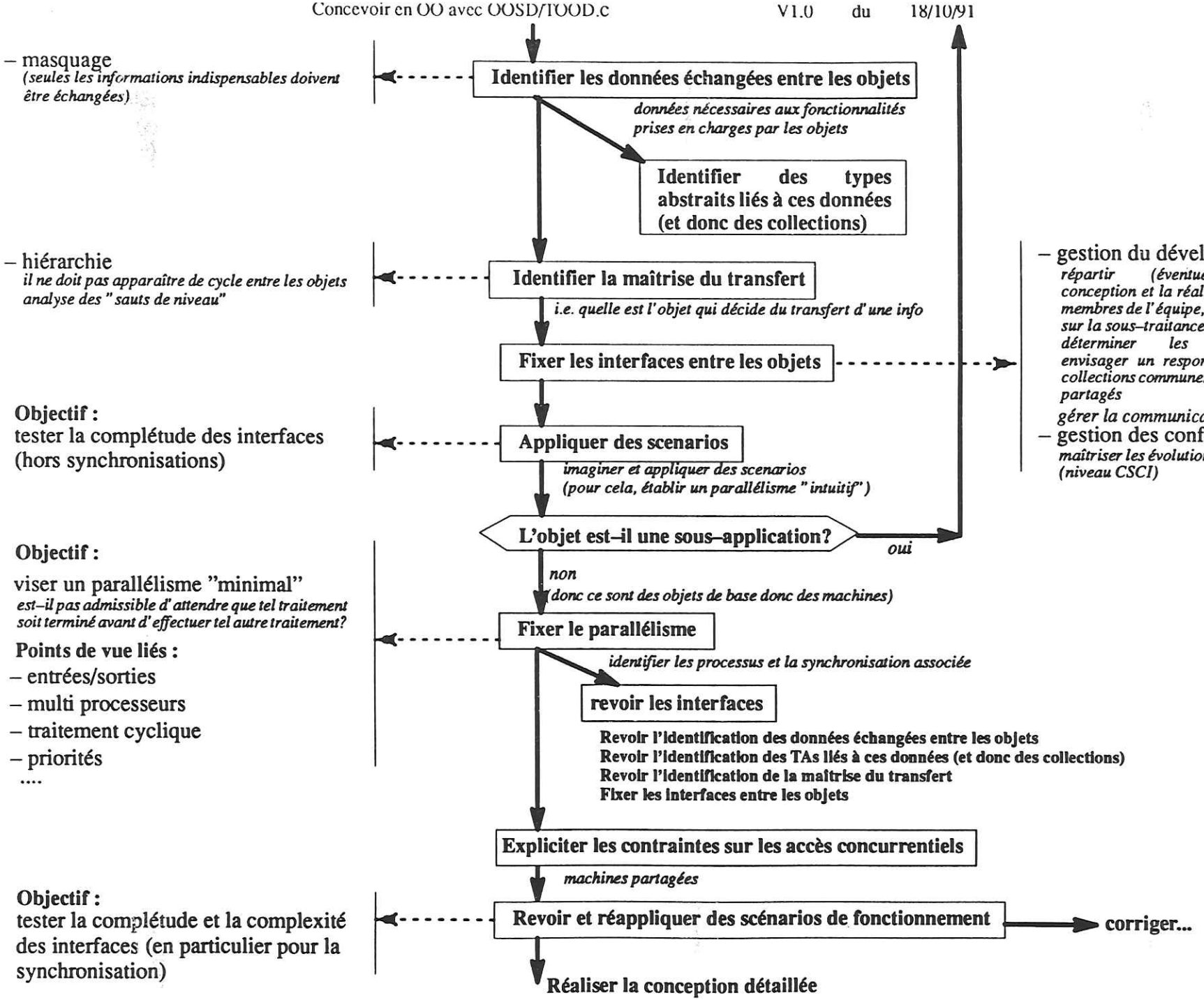


Figure 6.3. Vision synthétique de la démarche

Les paragraphes qui suivent présentent quelques conseils et des questions qu'il est bon de se poser (pas obligatoirement dans l'ordre présenté) sur les étapes identifiées.

6.4. Fixer l'interface de l'application

6.4.1. Objectif :

– description du mode de communication avec l'environnement

– description des messages échangés avec l'environnement

6.4.2. Guide :

1. Quelles sont les interactions reçues par ou émises vers l'environnement ?

> identification des messages/interruptions

2. Comment sont–elles définies ?

> identification des types de données des messages;

> identification des interruptions;

3. Comment la communication s'effectue–t–elle ?

3.1. Est–ce une communication "synchrone" (i.e. avec exigence de prise/émission immédiate par le système) *ou "asynchrone"* (i.e. avec possibilité de prise/émission avec retard par le système) ?

> identification des "priorités" de prise en compte, par le système, des messages (préambule à la question suivante); fréquence de réception/envoi ...

3.2. Faut–il stocker ces messages ? Combien ? Comment ?

> identification du mode de stockage (boîte aux lettres, processus de réception/stockage, ...), du fait qu'il doit ou non être à l'intérieur du système, de la taille nécessaire, identification des protections vis à vis de l'exclusion mutuelle sur ce stockage ...

4. Quels sont les contrôles à effectuer sur les messages reçus ?

> identification des contrôles de transmission, de l'exactitude, du dégré de précision, taux d'erreur, tous contrôles portant sur la sécurité (légalité des données, droits d'émission/réception, confidentialité,...)

5. Qui est maître de la communication (i.e. est–ce le système qui "appelle/envoie" les messages ou l'environnement lui impose–t–il de les "prendre/envoyer") ?

> identification du lieu (intérieur ou extérieur du système) des opérations de communication

6. Quelles sont les opérations de communication ?

> identification des opérations, des suites d'opérations pour une communication donnée (protocoles normalisés ou spécifiques, logique d'appel des opérations ...); écrire l'ordonnancement de ces opérations, à partir de toutes les informations collectées, dans une annotation de l'objet qui gère cette communication.

7. Quelles sont les principales informations intervenant dans le système, quelles sont les principales fonctions réalisées par le système, quelles sont les contraintes associées ?

> écrire un texte explicatif contenant ces éléments.

8. Quels sont les articles de configuration logiciels et/ou matériels avec lesquels l'application s'interface ?

> on utilise pour cela un diagramme de contexte appelé *diagramme de contexte opérationnel*.

9. Quels sont les logiciels pré–existants qui seront incorporés à la livraison ?

> Il s'agit des NDS (Non Developmental Software) de la DOD–STD–2167A, c'est à dire les logiciels réutilisés (Base de données, bibliothèques de composants, moniteur ... si ceux–ci ne sont pas identifiés comme articles de configuration).

Un diagramme présente les liens de l'application vers ces NDS. Ce diagramme est appelé *diagramme de contexte industriel*.

6.5. Identifier les candidats objets

6.5.1. Identification proprement dite des objets

6.5.1.1. Objectif :

– découvrir les candidats objets

6.5.1.2. Guide :

On rappelle également que le modèle d'expression des exigences n'est peut–être pas adapté à une reflexion sur la solution. Dans ce cas, le concepteur doit avant toute chose "reposer le problème" dans un modèle (mathématique, physique, relationnel ....) qui permettre de définir la solution logique.

De même si la solution logique est *très loin* de l'expression de la spécification, il peut être nécessaire de définir les grandes lignes de la solution logique, avant de s'attaquer à l'organisation du logiciel. Cependant, il faut savoir que l'organisation du logiciel s'appuyant sur la démarche proposée ici introduit une décomposition de l'application qui peut permettre de trouver cette solution complexe (comme étant constituée d'ensemble de solutions locales) qui aurait été introuvable sans cette décomposition *(et c'est le cas le plus fréquent!)*.

6.5.1.2.1. Applications de taille importante

Lorsque la spécification est un document très important, il est très pénible d'avoir une recherche systématique des objets.

Dans ce cas, plusieurs approches sont possibles :

– approche par capacité : se limiter à la partie de la spécification qui correspond à une capacité ou sous–capacité (et à sa décomposition);

– approche par "objets principaux" : repérer seulement les objets principaux de la spécification (fortement basé sur l'"intuition" et donc la compétence du domaine du concepteur);

– approche par niveau dans la spécification : c'est utilisable presque uniquement si la spécification est basée sur un modèle SADT ou SART. Dans ce cas, le concepteur part de l'hypothèse (pas toujours vérifiée) que les objets importants apparaissent dans les premiers niveaux des modèles de spécification.

On procède alors par enrichissement de la carte des objets, en faisant des compléments de lecture (analyse des objets d'une nouvelle fonction dans le premier cas, analyse des niveaux inférieurs dans le troisième cas) ou des relectures (découverte des objets moins importants dans le deuxième cas).

Si après une analyse de ce genre, il y a beaucoup, voire trop d'objets, il faut les regrouper en sous–applications (cf AUCUN LIEN).

6.5.1.2.2. A partir de SA–RT :

Les règles suivantes peuvent aider à découvrir des objets à partir d'un modèle SA–RT.

A) Ordre d'exploitation du modèle SA–RT

Il est en général plus simple d'exploiter d'abord le diagramme de contexte de données (DCD), puis les diagrammes de flots de données (DFD), puis les spécifications de processus et le dictionnaire de données.

Les diagrammes de contrôle (diagrammes de contexte de contrôle, DCC, et diagrammes de flots de contrôle, DCD), ainsi que les spécifications de contrôle (C–SPEC) font rarement apparaître des objets. On les utilise en général pour l'expression de l'enchaînement et en particulier pour la synchronisation, mais pas pour identifier les objets (sauf cas particulier). On les exploite donc en dernier.

B) Le Diagramme de Contexte (de Données)

B.a) les terminaisons

Dans le diagramme de contexte, les noms à l'intérieur des terminaisons sont de très bons candidats objets, car ils représentent souvent des entités matérielles (y compris l'utilisateur du logiciel) avec lesquelles le logiciel communique: le concepteur en fera une machine ou une sous–application, abstraction du matériel. Les noms représentent également parfois des logiciels: on les traite comme les matériels.

Un objet "attaché" à une terminaison regroupe les caractéristiques (utiles pour le logiciel) de l'entité matérielle : par exemple, états du matériel, paramètres, adresse mémoire ... Il peut contenir aussi un stockage des informations échangées avec ou via ce matériel. Il contiendra enfin les opérations de communication avec ce matériel. Note : les opérations (de communication) externes à l'objet sont les opérations vues du reste du logiciel; les opérations de base gérant la communication avec le matériel peuvent (et très probablement doivent) être masquées au reste du logiciel.

Une partie importante du parallélisme vient de la gestion des entrées–sorties. Pour gérer ces entrées–sorties, le concepteur sera amené à identifier des processus : la terminaison pourra alors être représentée comme une sous–application qui contiendra (lors d'une analyse ultérieure de cette sous–application) une ou plusieurs machines contenant toutes les informations (état, stockages, paramètres ...) et un ou plusieurs processus qui piloteront les entrées–sorties en s'appuyant sur ces machines.

Remarque 1 : les terminaisons peuvent être des regroupements logiques (fonctionnels) de matériels et/ou logiciels avec lesquels le logiciel communique. Dans ce cas, on devrait trouver dans un commentaire associé au diagramme de contexte l'identification détaillée de chacun de ces composants de la terminaison. L'identification des objets s'appuiera alors sur chacun de ces composants, surtout si les protocoles (au sens général du mot, c'est–à–dire pas obligatoirement protocole normalisé) sont différents.

Remarque 2 : Ces objets "d'interface" sont une bonne manière d'obtenir une bonne portabilité en isolant ainsi les aspects d'interfaçage.

B.b) les flots de données

Les données qui apparaissent sur les flots de données sont des informations qui vont probablement se "déplacer" dans l'application (paramètres d'opérations), et/ou qui existeront en plusieurs exemplaires dans l'application; elles seront modélisées par l'intermédiaire de types abstraits (de données) dont elles seront des instances.

Remarque : attention, les flots de données qui apparaissent dans le diagramme de contexte peuvent être des noms donnés à des regroupements (logiques ou non) de données : ces regroupements ne seront peut–être pas maintenus dans le logiciel, dans lequel on manipulera uniquement les données "de base" ou plus exactement les données qui sont les paramètres des opérations effectives de communication avec l'extérieur.

B.c) le processus 0

Le nom situé dans le processus 0 peut parfois identifier un objet. Dans ce cas ce sera un objet important.

Attention, ce nom peut parfois être tellement général qu'il n'apporte pas un objet précis mais un ensemble : le concepteur pourra alors soit identifier une sous–application pour le représenter, soit l'ignorer purement et simplement, car il est trop général et ses composants seront de toute façon identifiés plus tard.

Le nom situé dans le processus 0 peut aussi identifier un type d'objets lorsque qu'il désigne un ensemble d'informations traitées par le logiciel (par exemple : gérer les pistes...).

B.d) exemple partiel de lecture d'un diagramme de contexte

Sur le diagramme de contexte suivant, on a fait apparaître les deux aspects données et contrôle.

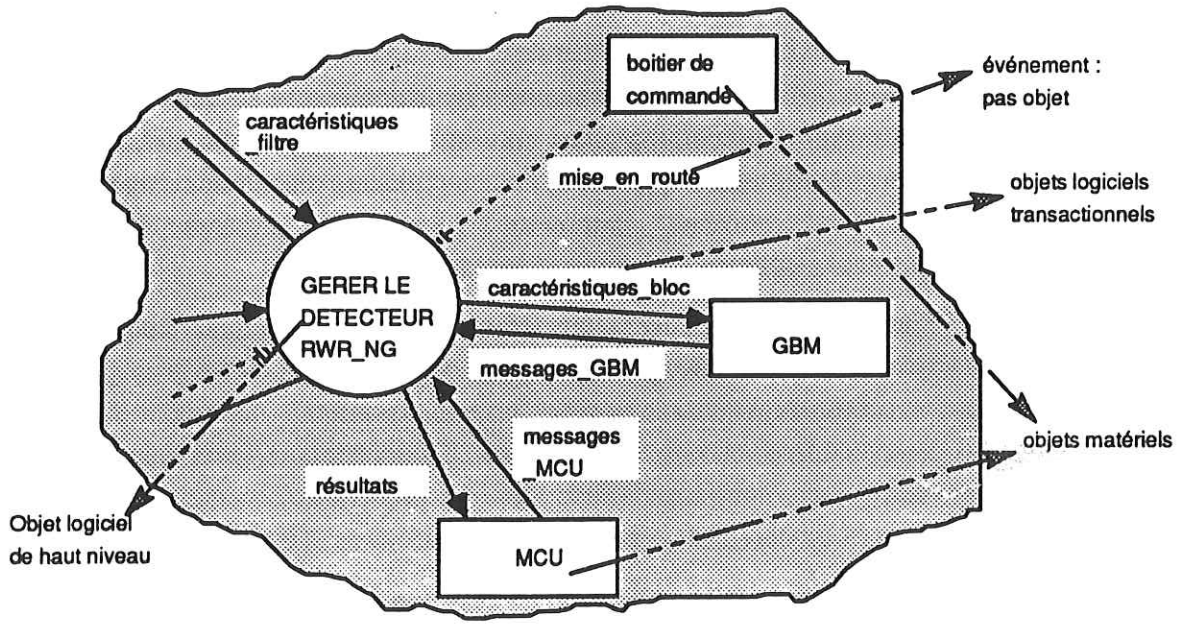


Figure 6.4.

C) Les Diagrammes de Flot de Données

C.a) Les stockages

Un stockage de SA–RT est en général traduit

- soit comme un objet à part entière (machine);
- soit comme un attribut d'un autre objet (représenté par une machine en général); c'est toujours le cas pour les stockages qui désignent des paramètres de l'application; c'est également le cas, lorsque le concepteur considère que l'information gérée par le stockage est une partie de l'information associée à un objet déjà identifié.

Ces stockages peuvent de plus faire apparaître un type abstrait pour modéliser ce qui se trouve dans le stockage. La machine est une représentation du contenant (la table, le buffer ...), alors que le type abstrait est une modélisation du contenu.

Par exemple, dans la figure 6.5. qui suit, on pourra trouver une machine *TableDesParamètresAvion*, correspondant au stockage des *ParamètresAvion*, dont chacun des éléments sera défini comme une instance d'un type abstrait *Paramètre_Avion*.

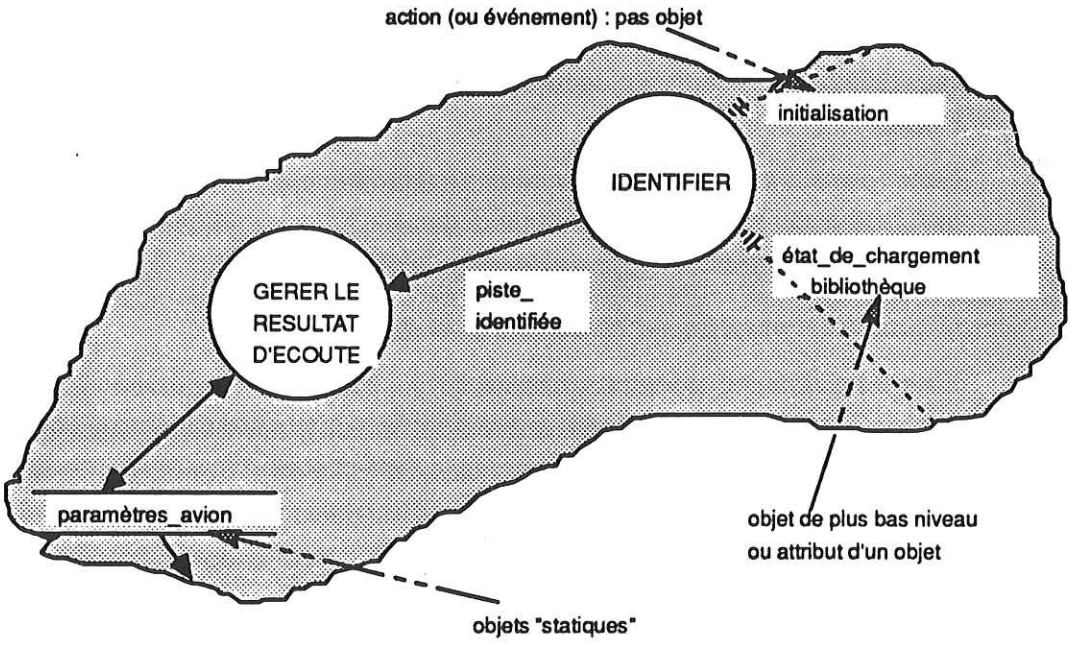


Figure 6.5.

C.b) Noms dans les processus

Dans les diagrammes de flots de données, les noms qui apparaissent dans les processus sont de bons candidats objets. Ils peuvent être traduits en machines ou en TADs. Pour un grand nombre d'entre eux, on pourra les retrouver soit sur un flot de données, soit comme identificateur d'un stockage.

C.c) Les flots de données

Comme dans le diagramme de contexte, les données qui apparaissent sur les flots de données seront modélisées par l'intermédiaire de types abstraits dont elles seront des instances.

La remarque les concernant dans le diagramme de contexte peut s'appliquer aux flots de données des premiers niveaux de décomposition des diagrammes de flot de données.

De plus, si un flot de données apparaît uniquement entre deux processus dont les transformations qu'ils spécifient sont prises en charge par un même objet, le flot de données sera local à l'objet en question. Il n'est alors pas du tout évident que le flot de données serqa "traduit" en un type abstrait.

C.d) Note sur les contrôles

Il faut noter que certains contrôles, s'ils ne désignent pas d'objets, peuvent faire apparaître des objets sous–jacents.

Dans l'exemple donné dans la figure 6.5. ci–dessus, on voit apparaître le contrôle *état_de_chargement_bibliothèque* :

- cela évoque donc le fait qu'il existe une bibliothèque. Or cette bibliothèque n'apparaît pas dans les diagrammes, car elle est considérée comme locale au processus "identifier";
- de même, l'état de chargement est une information qui sera probablement utilisée; cet état sera probablement un attribut de la bibliothèque.

Le contrôle n'est donc pas "directement" traduit en un objet, mais il a alors servi à faire apparaître une information modélisée par un objet (et même dans ce cas, un attribut de l'objet).

Note : normalement, la spécification devrait faire apparaître l'objet par ailleurs (mais il faut faire avec la spécification dont on dispose!). Ici, en principe, la P_SPEC associée à "identifier" devrait faire apparaître cette bibliothèque.

D) Spécifications de processus et dictionnaire de données

Toutes les données qui sont apparues sur les diagrammes doivent se retrouver dans le dictionnaire de données. Il se peut que la définition d'une donnée introduise de nouvelles données de plus bas niveau : dans ce cas, celles–ci sont des candidats objets (le fait de savoir si ce sont de "bons" candidats dépend du niveau de détail auquel elles se situent). Il se peut que des commentaires divers leur soient associés : le concepteur se retrouve alors dans un cas d'analyse d'un texte (cf 6.5.1.2.3.).

La spécification d'un processus (exprimée en texte libre ou en texte semi formel du genre pseudo–langage) peut contenir de nouveaux noms et donc introduire des candidats objets. En particulier, certaines constantes peuvent apparaître dans la P–SPEC (par exemple la bibliothèque de la figure 6.5. sera présente dans la P–SPEC associée au processus Identifier).

Note : il est parfois utile de gérer une information d'état correspondant à un automate de C_SPEC. Dans ce cas, elle apparaît comme attribut de l'objet qui "pilote" la capacité spécifiée par le processus (auquel cette C–SPEC est associée).

E) S'assurer que ces candidats objets sont de bons candidats

Les questions suivantes développées en 6.5.1.2.3. peuvent servir de "check–list" pour éliminer les "faux objets".

6.5.1.2.3. Cas des spécifications textuelles

1. Quels sont les noms *qui apparaissent dans la spécification ?*

> A partir d'une spécification textuelle, cela consiste à relever tous les noms qui apparaissent dans la spécification. En effet, chacun des noms est susceptible de représenter un objet. On parlera de candidats objets.

2. Ces noms représentent–ils des actions ?

> Le but est d'éliminer les noms qui évoquent des actions (c'est le cas en particulier de presque tous les mots dont le suffixe est "ation").

3. Ce nom représente–t–il un objet du monde réel ?

> Le but est d'avoir une tendance à limiter le nombre d'objets.

4. Peut–on associer un état à l'entité identifiée par ce nom ? Quelles sont ses caractéristiques ?

> Le but est
>
> – d'une part d'éliminer toute entité qui ne possède pas d'état ou d'attribut, car elle ne peut représenter un objet.
> – d'autre part, de préciser ce que cet objet représente comme "valeur informative" pour le logiciel.

5. Peut–on associer l'entité à un autre objet ? L'entité est–elle une partie d'un (autre) objet ?

> Le but est de limiter le nombre d'objets identifiés.

6. Y a–t–il des synonymes ?

> Le but est d'éviter de disperser les informations.

7. Quels sont les noms qui correspondent à des valeurs (figées)?

> Le but est d'identifier les constantes, les ensembles logiques de valeurs (traduits en langage de codage par des types énumératifs).

8. Quels sont les objets qui ne seront pas représentés dans l'application ?

> Le but est de "dégraisser" l'application.

6.5.2. Identification des fonctionnalités

6.5.2.1. Objectif :

– identifier les fonctionnalités à réaliser,

– associer aux objets ces fonctionnalités.

6.5.2.2. Guide :

6.5.2.2.1. Introduction

Une action dans la spécification peut représenter une "fonctionnalité" dont la réalisation se traduira par une ou plusieurs opérations sur le même objet ou sur des objets différents.

Dans le cas ou une fonctionnalité est associable à plusieurs objets, on associera la responsabilité de la fonctionnalité à l'un des objets, c'est à dire qu'il pilotera la fonctionnalité. En particulier pour les fonctionnalités de haut niveau, la responsabilité pourra être donnée à un (ou plusieurs) processus (de conception cette fois!).

6.5.2.2.2. A partir de SA–RT

A) Ordre d'exploitation du modèle SA–RT

Comme pour l'identification des objets, il est en général plus simple d'exploiter d'abord le diagramme de contexte de données (DCD), puis les diagrammes de flots de données (DFD), puis les spécifications de processus.

Mais, contrairement à l'identification des objets, on exploite également les diagrammes de contrôle (diagrammes de contexte de contrôle, DCC, et diagrammes de flots de contrôle, DCD), ainsi que les spécifications de contrôle (C–SPEC).

B) Les Diagrammes de flots

Les verbes situés dans les bulles "processus" sont de bons candidat objets. Les verbes situés dans les diagrammes de haut niveau peuvent être vus comme des "ordonnanceurs" partiels (i.e. d'une partie de l'application). Il se peut qu'on ne sache pas à qui les attribuer tellement ils utilisent d'objets. Ils seront en général associés à des entités sous–applications et seront "gardés sous le coude" pour être utilisés lors de l'analyse de la couche "pilotage" de l'application (en particulier dans un contexte parallèle) et seront probablement associés aux processus.

C) Les P–SPEC

Dans les P–SPEC, on peut trouver un certain nombre d'opérations (sous forme de verbes), en général de bas niveau ... fonctionnellement parlant, car elles peuvent être non simples.

D) Les C–SPEC

Il peut être utile d'associer des opérations aux actions associées aux transitions de l'automate dans la C–SPEC, surtout si le nombre d'activations/désactivations/émission de contrôles est important.

E) Les contrôles

Les contrôles peuvent évoquer des opérations, c'est ce qui explique l'intéret pour tout ce qui touche aux contrôles.

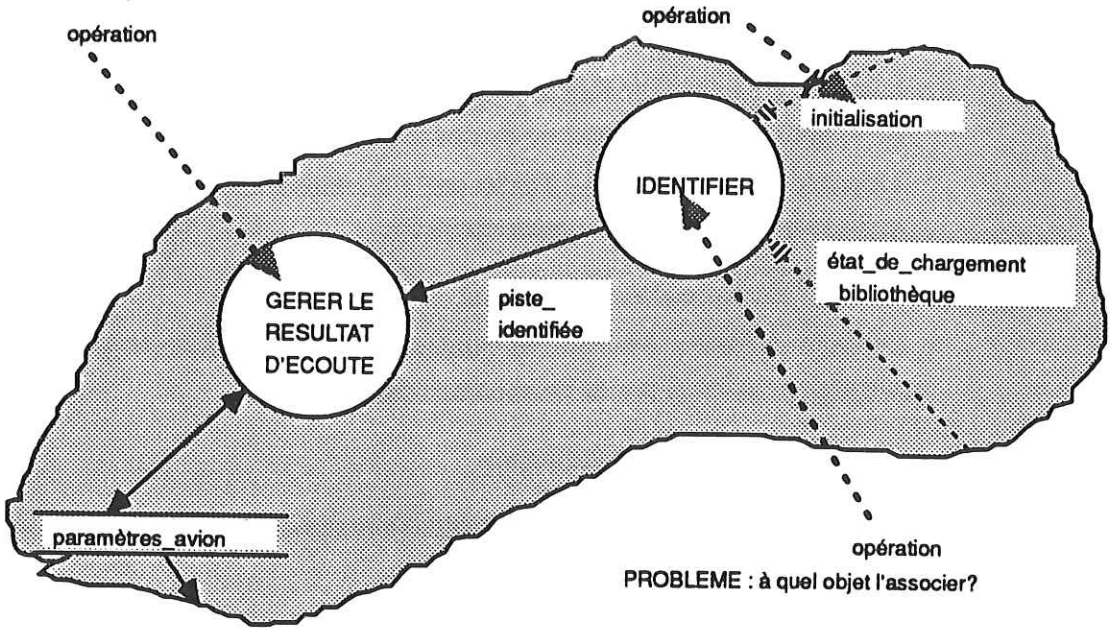


Figure 6.6. Diagramme de flot de données SA–RT

Ainsi dans l'exemple de la figure 6.6. :

- *initialisation* est un événement qui évoque le fait qu'il existe une opération initialiser,
- *état_de_chargement* introduit une possible opération de consultation de cet état.

F) Le dictionnaire de données

Le dictionnaire de données n'introduit pas ou peu d'opérations. Cependant, une analyse de la structure de données peut évoquer des opérations (de modification ou de déplacement dans la structure en particulier). Certaines informations sur les contraintes peuvent également évoquer des opérations de contrôle ou d'interrogation.

6.5.2.2.3. *A quel objet associer l'action ?*

On essaie d'associer, en général, dans l'ordre ci–dessous, l'action à l'objet associé

- soit au nom situé dans le même processus (complément d'objet du verbe), soit à l'un des stockages liés en entrée ou en sortie au processus – cas fréquent;
- sinon, à l'une des données correspondant à un flot d'entrée – cas possible;
- sinon, à l'une des données correspondant à un flot de sortie – cas rare.
- sinon, un objet de stockage de données d'entrée doit être probablement identifié (cf 6.5.3.2., question 8.), et l'opération lui sera alors associé – cas raisonnablement courant.

6.5.2.2.4. A partir d'une spécification textuelle

1. Quels sont les verbes *qui apparaissent dans la spécification ?*

2. A quel objet doivent–elles être associées ?

> Une action est associée à un objet seulement si elle est une manipulation de cet objet (consultation, modification, suppression ... et avec d'extrèmes réserves : création, la création ne pouvant être appliquée à l'objet, puisqu'il n'existe pas encore, mais étant plutôt une modification du contexte d'appel de cette création).
>
> L'opération est associée à l'objet qui a la connaissance nécessaire (pas toujours suffisante) pour effectuer cette opération.

3. Que faut–il faire d'autre sur ces objets ?

> Il s'agit d'étudier les besoins potentiels de manipulation de l'objet en terme d'opérations, en plus des besoins identifiés.

6.5.3. Complément d'analyse sur les objets

Les choix d'identification des types abstraits de données et des machines sont effectués à la suite de ce complément d'analyse.

6.5.3.1. Objectif :

– identifier les éléments de réutilisation

– associer aux objets tout attribut utile (état, structure, ...).

– découvrir et affecter les contraintes sur les objets.

– mettre à jour les besoins de "mémorisation" des objets par l'application.

6.5.3.2. Guide :

1. Cet objet ressemble–t–il(dans son comportement et/ou dans son contenu) à un autre objet, soit de l'application, soit existant dans une bibliothèque de composants réutilisables ?

> le but est
>
> – de factoriser (regrouper) tout ou partie des objets utilisés à plusieurs reprises dans l'application,
>
> – de ne pas redévelopper l'existant, mais de le réutiliser.

2. Cet objet a–t–il un comportement (général) réutilisable dans d'autres projets, i.e. est–ce un objet " classique" dans les logiciels du domaine de compétence de l'entreprise ?

> le but est d'identifier des logiciels qui pourront être réutilisables, donc de les développer avec un soin particulier, afin qu'ils aillent enrichir la bibliothèque des composants réutilisables.

3. Comment caractériser davantage les objets ? i.e. quels sont les qualificatifs (en particulier adjectifs) sur ces objets ?

4. Quelles sont les contraintes qui apparaissent dans la spécification ?

5. Comment ces contraintes sur l'objet vont–elles se propager sur les autres objets?

> Il s'agit de savoir si ces contraintes à propager doivent porter sur la définition même des autres objets ou si c'est seulement l'utilisation spécifique de ces objets par l'objet considéré qui est contrainte.

6. Quelle est la durée de vie de l'information portée par l'objet?

> L'information portée par l'objet peut être *rémanente* (c'est à dire pertinente et exploitable pendant un temps donné non–instantané) ou *volatile* (résultat d'une action, elle est utilisée pour l'exécution d'une ou plusieurs autres actions et disparaît ensuite). Il est possible de préciser ce point par les questions suivantes:

6.1. L'information correspondant à cet objet est–elle persistante ou volatile ?

6.2. L'information correspondant à cet objet est–elle "maintenue" pour une ou plusieurs actions ?

7. Quelle est la durée de vie de l'objet lui–même?

> Il s'agit de bien différencier (s'il y a lieu) l'information portée par l'objet de l'objet lui–même (contenu et contenant). Par exemple, un objet peut être une zone permanente d'information, alors que l'information localisée dans cette zone est volatile ou à durée de vie très limitée.

7.1. Peut–on envisager de ne pas considérer cet objet comme entité logique, mais seulement comme une valeur transmise entre deux actions ?

8. Quels sont les besoins de stockage de l'objet ?

> Il se peut qu'à un ou plusieurs points de l'application, un certain nombre d'informations de même structure et de même sémantique (signification) soient reçues pour être traitées. Le concepteur doit se préoccuper de la sauvegarde de ces informations s'il ne peut assurer qu'elles seront traitées "au fil de l'eau".

8.1. Une suite d'objets peut–elle être transmise ? Peut–il y avoir plusieurs points d'accumulation de ces objets ? Faut–il sauvegarder cette suite ?

9. Quelles sont les contraintes sur l'utilisation de l'objet ?

> le but est de définir la séquence d'appels d'opérations autorisable.

Cette "séquence" contient les contraintes sur l'accès concurrentiel à l'objet ou à tout ou partie de ses opérations.

6.5.4. Complément d'analyse sur les fonctionnalités

6.5.4.1. Objectif :

- décrire le rôle des fonctionnalités,
- identifier les objets et leurs opérations qui vont intervenir dans la réalisation des fonctionnalités.
- en particulier, identifier les opérations (externes) de l'objets qui permettent "d'utiliser" ces fonctionalités

6.5.4.2. Guide :

Lorsque le transfert de la spécification vers la conception est terminé, associer aux opérations présentes (issues de l'identification des opérations des passages précédents dans le cycle d'étapes), en partant des opérations de haut niveau (i.e. celles identifiées comme fonctions de base; pour SADT ou SA–RT, fonctions identifiées dans les premiers niveaux) jusqu'aux opérations de bas niveau (i.e. les feuilles dans l'arborescence SADT ou SA–RT), un commentaire sur le service qu'elles rendent (c'est leur rôle : vision boîte noire) et comment, grossièrement, elles le font, c'est–à–dire sur ce qu'elles seront amenées à faire ou à "faire faire" pour remplir leur rôle.

Pour les opérations (fonctionnalités) de haut niveau (apparaissant dans les premiers niveaux de décomposition de la spécification), la spécification peut l'exprimer de manière suffisante, et le concepteur peut réutiliser ce qui a déjà été décrit. Pour SA–RT ou SADT, on peut s'appuyer sur la décomposition en processus.

Pour les opérations de plus bas niveau, la spécification a introduit une définition qui peut difficilement être "récupérée en l'état" pour la conception. Dans ce cas, le concepteur doit réfléchir au "comment" réaliser cette opération, et décrire la stratégie informelle de résolution.

C'est le moment pour le concepteur de faire intervenir son imagination. Il/elle va envisager différents scénarios de solution, et en faire l'analyse pour l'identification des opérations. Cette activité est peu formalisée et sa réussite dépend fortement de l'expérience du concepteur.

6.6. Répartir les candidats–objets en entités "nuages"

Le but visé est d'obtenir un nombre "d'objets" raisonnable pour maîtriser la complexité (en réalisation et en relecture) de la "carte des objets", à un niveau donné.

Chaque sous–application sera alors développée comme une "mini" application sur laquelle le concepteur mènera une reflexion approfondie conduisant à l'identification des objets et de leurs relations.

Bien que ce soit rare, il se peut, qu'après cette reflexion, le concepteur se rende compte que ce qu'il avait considéré comme une sous–application contenant plusieurs candidats objets, se réduise en fait à un seul objet comprenant des attributs correspondant aux candidats objets.

Il peut être judicieux de faire plusieurs essais de répartition, et d'aller jusqu'à fixer les interfaces afin d'évaluer les différentes alternatives.

6.6.1. Objectif :

– identifier les objets composites (sous–applications ou objets non–stricts)

– répartir les objets ou candidats objets entre ces objets composites, et donc répartir leurs "fonctionnalités".

6.6.2. Guide :

Il est toujours possible de regrouper, mais il est utile d'identifier des "critères" de regroupement et d'évaluer les avantages et inconvénients de ces critères.

La répartition des objets dans les sous–applications se fait à partir de considérations de couplage et de cohésion: les objets regroupés dans une même sous–application doivent être amenés à fortement "travailler" ensemble.

Les principaux critères sont les suivants :

– *regroupement fonctionnel*

Remarque : il ne s'agit pas pour autant d'une approche fonctionnelle, puisque les objets sont d'abord identifiés indépendamment de la ou les fonctions qui les utilisent.

– *regroupement lié à l'organisation du réel*

Dans le monde réel, lorsque plusieurs éléments sont regroupés physiquement (en participant ou non aux mêmes fonctions), ils constituent un assemblage qui est en lui–même un élément.

Un regroupement lié à l'organisation du réel a lieu lorsque le regroupement d'objets contient les objets qui sont chacun l'image informatique d'un composant de cet assemblage.

– *regroupement lié au domaine de compétence des concepteurs/réalisateurs*

Ce n'est plus un critère technique sur le produit, mais un critère technique sur le développement du produit : c'est un argument purement industriel. En effet, dans un projet, les concepteurs/réalisateurs ne peuvent être compétents sur tous les domaines (graphisme, communication, BdD ...). Il existe des spécialistes à qui il s'agit de confier une partie bien identifiée du logiciel.

Un regroupement lié au domaine de compétence a lieu lorsque le développement des objets du regroupement nécessite une compétence particulière sur un domaine donné.

– *regroupement lié à la facilité de réaliser l'intégration*

Regrouper des objets dans des sous–applications de telle sorte que l'on voit apparaître des cycles d'appels entre sous–applications est dangereux pour l'intégration. On pourra donc changer ces regroupements ou changer la responsabilité (attribuée à tel ou tel objet) des relations entre les objets répartis dans les sous–applications.

Cela comprend également des considérations industrielles telles que le coût ou la spécificité du banc de test, la possibilité de définir une stratégie d'intégration raisonnable. Ces considérations sont utilisées plutôt comme des "checklist" de la conception.

– *regroupement lié aux interfaces*

Il a lieu lorsque les objets du regroupement se trouvent là parce qu'ils participent à la communication avec "l'extérieur" du logiciel. C'est le cas lorsque l'on veut "banaliser" une interface.

Pour les collections, le regroupement logique (regrouper des types abstraits s'ils ont des natures équivallentes, donc s'ils modélisent des entités qui rendent le même genre de services) est aussi possible.

6.7. Fixer l'interface des sous–applications

Dans le schéma général de la démarche, les trois étapes : identifier des données échangées entre les objets, identifier la maîtrise du transfert et fixer les interfaces entre les objets, conduisent à la dernière (fixer les interfaces). Les commentaires sont regroupés ici.

6.7.1. Objectif :

- déterminer les liens entre les objets
- définir une bonne visibilité entre composants (la plus faible possible) et, dans un deuxième temps, mettre en place, si possible, une hiérarchie
- description du mode de communication entre objets composites (sous–applications ou objets non–stricts)
- définition des messages échangés entre objets composites (sous–applications ou objets non–stricts)

6.7.2. Guide :

1. Quelles sont les informations échangées entre les objets composites ?

> identification des données

2. Peut–on identifier les opérations de transfert ?

> dans la mesure du possible, il faut nommer ces opérations par un nom qui évoque le service rendu ou la fonctionnalité "enclenchée", et pas seulement par un nom de transfert (recevoir, émettre ...).

3. Quel est la sous–application maître de la communication (qui décidera quand se fera cette communication, et donc appelera l'opération) et celui qui est esclave (qui contiendra l'opération de transfert)?

4. Qui est appelé par qui ?

> à la suite du complément d'analyse sur les fonctionnalités une dépendance d'appels peut être établie : si une opération d'un objet A est appelée par un objet B, un lien de visibilité de B vers A sera établi.

5. Tel objet a–t–il vraiment besoin de voir tel autre objet ?

> entre autre, si le niveau de détail entre deux objets est très différent, il est probable qu'ils ne doivent pas se voir directement. Dans ce cas, il faut supprimer ce lien (peut–être en en créant d'autres ou en utilisant des liens déjà existants).

6. Y a–t–il des cycles de maîtrise des échanges ?

> les cycles sont toujours un problème pour l'intégration et le test: il faut donc les éviter.

> Pour cela, s'il existe des cycles, on essaie de les "casser", soit en inversant certaine(s) responsabilité(s), soit en transmettant l'information par un autre moyen (par passage de paramètre du maître vers l'esclave, au lien d'attendre que l'esclave demande cette information et donc introduise un cycle).

7. Y–a–t–il des sauts de niveau?

> Un saut de niveau peut révéler un mélange de niveaux d'abstraction (ou niveaux de détail) pour un objet donné, ce qui, si c'est le cas, est à proscrire.

supprimer ou réduire les sauts de niveaux; justifier ceux qui restent.

Cette activité peut conduire à une remise en cause des choix de répartition des informations dans les différentes entités.

8. Quelle sont la logique (protocole) d'échange et les contraintes?

9. Ou sont instanciés les types abstraits ?

> si un objet ou type abstrait A contient une ou plusieurs instances d'un type abstrait B, un lien de visibilité sera établi de A vers B (et éventuellement vers la collection qui contient le type abstrait).

10. Lorsque le parallélisme est défini, et qu'il y a révision des interfaces : Quel est le mode de communication (synchrone, asynchrone) ? S'il y a une boîte aux lettres, une FIFO ..., dans quelle sous–application sera–t–elle placée ?

6.8. Analyser l'ordonnancement (et donc fixer le parallélisme)

Rappel : les processus sont avant tout des pilotes. Il faut éviter au maximum de leur faire "faire un travail" autre que le pilotage et/ou la synchronisation.

La conséquence de cette activité va être la remise à niveau des interfaces, au moins pour tout ce qui touche aux synchronisations. La criticité des machines en terme de partage va également être établie. Il faudra donc revoir les machines partagées pour préciser les contraintes d'accès concurrentiel.

6.8.1. Objectif :

- identifier l'ordonnancement des actions et les besoins en parallélisme entre les actions.
- en déduire l'identification et la répartition des processus.

6.8.2. Guide :

1. Les actions identifiées doivent–elles être parallélisées ?

> L'objectif n'est surtout pas de chercher du parallélisme à tout prix, mais au contraire de déterminer un parallélisme si possible minimal.
>
> Le parallélisme doit être introduit quand il n'est pas admissible d'attendre qu'une action (ou qu'un ensemble d'actions) se termine avant de lancer une autre action.

2. Les actions identifiées peuvent–elles être parallélisées ?

3. Dans quel ordre s' effectuent les actions ? Quels sont les modes de communication/synchronisation (procédural, synchrone, asynchrone ...)?

> Il est préférable, lorsqu'on a identifié du parallélisme, d'avoir une idée de la synchronisation d'ensemble, pour voir "si ça peut marcher". Il est conseillé alors d'envisager des scénarios de fonctionnement.

4. Où répartir les processus ?

> il s'agit d'identifier les sous–applications dans lesquelles les processus vont être placés. Le processus est placé dans la sous–application qui contient les machines abstraites que le processus pilote. Si le processus pilote des machines abstraites réparties dans plusieurs sous–applications, l'une des sous–application est choisie pour "piloter" tout ou partie des autres sous–applications et le processus est alors placé dans cette sous–application.

La détermination du parallélisme (donc des processus) peut introduire une retouche des objets pilotés ou exploités par les processus (nouvelles opérations d'interrogation, introduction d'attributs genre "état", voire sémaphores de protection ...).

6.8.3. A partir de SA–RT :

Les C–SPEC (specifications de contrôle) peuvent être exploitées pour identifier l'ordonnancement des opérations (correspondant aux processus de SA–RT) prises en charge par les objets.

L'analyse des flots de données permet également de déduire un séquencement des opérations.

Remarque importante : Attention à la signification du "parallélisme en SA–RT : il indique seulement (sauf indication contraire expressément indiquée ... qui semblerait surprenante dans une spécification des exigences servant à la relation client/réalisateur) que les fonctions correspondant aux différents processus "déclenchés" simultanément à partir d'un événement sont "déclenchables" (peuvent commencer à partir de ce moment) à partir de cet événement.

6.8.4. Genres classiques de processus

On peut utiliser des règles pour un certain nombre de genres de processus qui apparaissent classiquement dans une application :

– *Processus de gestion des entrées–sorties*

– *Criticité par rapport au temps*

certaines fonctions doivent être effectuées dans un délai de temps serré, et souvent il est nécessaire de leur attribuer un processus propre (de haute priorité).

A l'opposé, les transformations qui ne sont pas critiques en temps (et qui représentent des calculs importants) peuvent être pilotées par des processus "tâches de fond" (avec une faible priorité) qui consommera le temps CPU restant.

– *Cohésion temporelle*

Les transformations que réalisent certaines fonctions durant la même période de temps, ou immédiatement à la suite de certains événements, peuvent être pilotées en un seul processus.

– *Fonctions périodiques*

Les fonctions périodiques doivent être implémentées en processus disjoints qui sont activés aux intervalles de temps qui leur sont propres.

De telles fonctions ne doivent pas être combinées en un seul processus, parce qu'il serait assez difficile de programmer plus d'une fonction périodique dans les conditions de suspension/reprise du processus.

6.9. Choix de la description des informations (machine ou type abstrait?)

Cette activité n'est pas présentée dans le synoptique car elle peut intervenir à tout moment, être réalisée plus ou moins partiellement à un moment donné, et il est donc difficile de la situer dans un ordonnancement.

6.9.1. Objectif :

– déterminer si les informations vont être décrites par des types abstraits ou par des machines.

6.9.2. Guide :

Il s'agit de définir les types abstraits et les machines, l'information qu'elles gèrent.

On rappelle qu'il faut voir un type abstrait comme un modèle d'information décrivant en général un contenu et la machine comme un contenant (avec son contenu).

1. L'information X doit–elle être partagée entre plusieurs processus?

> Si oui, l'information sera encapsulée dans une machine (se poser le problème de la nécessité d'une protection).

2. L'information X doit–elle utiliser une ressource encapsulée dans une machine (donc faire appel aux opérations d'une machine)?

> Si oui, l'information sera encapsulée dans une machine;

3. Existe–t–il plusieurs instances de cette information ?

> si oui, elle sera probablement décrite par un type abstrait. Attention, les instances doivent apparaître quelque part, surtout si elles sont partagées.

4. L'information apparaît–elle comme paramètre d'une opération ?

> si oui, elle sera très probablement décrite par un type abstrait.

5. L'information X doit–elle être avoir la durée de vie de l'application?

6. L'information X doit–elle être à priori ignorée par le processus (de trop bas niveau de détail par rapport au processus)?

> Si les réponses aux deux questions précédentes sont positives, l'information sera encapsulée dans une machine.

Dans les autres cas, c'est un choix de conception : a priori, il peut paraître plus simple d'introduire un type abstrait :

– si l'information doit avoir la durée de vie de l'application, elle sera locale à un processus abstrait ou à une machine et "héritera" donc de leur durée de vie.
– si l'information n'est présente que pour la réalisation d'une action, la donnée instance du type abstrait sera locale à l'opération qui réalise cette action.

7. Où mettre telle information (qui contient qui) ?

> On rappelle qu'une information peut être une ou plusieurs données (instances d'un type abstrait) ou une abstraction d'une entité matérielle.
>
> Il est possible de regrouper soit des information de même genre (cas d'un stockage de données reçues/émises), soit des informations de genres différents.

Note : in fine, une information doit apparaître dans le "plan de l'application" i.e. dans un processus ou une machine. Si l'information a été décrite par un type abstrait, une/des instances apparai(sen)t dans un/des machines ou processus.

7. PRINCIPES DE CONCEPTION

Les principes abordés sont ceux présentés également par G. BOOCH [BOOCH–91].

7.1. Décomposition / *Decomposition*

Processus qui permet de décrire une entité comme étant constituée de sous–entités, et définissant les inter–dépendances entre ces entités.

C'est ce processus qui permet d'aborder un problème complexe en le décomposant en sous–problèmes. Voir le discours sur la méthode (Descartes) ... ou encore "divide et impera" (diviser pour régner) de ...

Dans l'approche orientée par les objets, le logiciel est composé d'objets s'utilisant les uns les autres. Certains objets étant complexes, ils se décomposent en (sous–)objets.

7.2. Modularité / *Modularity*

Définition 1 :
Les caractéristiques du logiciel qui présentent une structure en modules[1] très indépendants les uns des autres. [VQ 02–1990 E/F]

Définition 2 :
The extent to which software is composed of discrete components[2] such that a change to one component has minimal impact on other components. [ANSI/IEEE 729–1983]

La modularité est donc directement liée au processus de décomposition du logiciel.

L'aspect d'indépendance entre les entités est précisé par deux concepts :

le couplage : il caractérise le niveau de dépendance entre entités;
un fort couplage a lieu entre deux entités lorsqu'elles sont fortement dépendantes,
un couplage est nul lorsqu'elles sont indépendantes;
le couplage doit donc être minimal;

la cohésion : il caractérise le niveau de dépendance entre les composants d'une entité (ces composants peuvent être ou non des entités);
une forte cohésion a lieu lorsque ces composants sont fortement dépendants; une faible cohésion a lieu par exemple lorsque deux sous–ensembles de composants d'une entité sont indépendants;
la cohésion doit être maximale;

L'intéret de la modularité est

– de faciliter la mise au point (en réduisant le développement lié aux entités pilotes (drivers) et muets (stubs), en facilitant la localisation des erreurs),

1. le terme entité de ce document est synonyme à cette notion de module
2. le terme entité est synonyme à cette notion de composant

– de réduire les coûts de maintenance (en facilitant la modification, puisque l'impact est bien localisé),

– d'être un préalable à toute politique de réutilisation (un composant fortement dépendant de son contexte n'est pas réutilisable, sauf exception).

7.3. Abstraction / *Abstraction*

Le sens commun, donné par le Larousse, est le suivant : "l'abstraction est l'opération intellectuelle qui permet d'isoler une caractéristique, une propriété d'un objet et de la considérer indépendamment des autres".

Cette définition peut être étendue ainsi : " l'abstraction est l'opération intellectuelle qui permet de définir un objet par l'ensemble des propriétés qui le caractérisent dans un domaine donné".

(1) A view of a problem that extracts the essential information relevant to a particular purpose and ignores the remainder of the information.
(2) The process of forming an abstraction. [ANSI/IEEE 729–1983]

La notion de domaine correspond à la notion de point de vue.

Définition 3 : An abstraction denotes the essential characteristics of an objet that distinguish it from all other kinds of objets and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.[3] [BOOCH–91]

A la notion d'abstraction est directement liée la notion de niveau de détail. En effet, pour aborder la résolution d'un problème (et pour la présenter), il n'est pas nécessaire (et il est la plupart du temps dangereux) de mettre au même plan tous les niveaux de détail sur le problème et de vouloir les résoudre tous en même temps. Par contre, bien entendu, la résolution complète du problème suppose la résolution de tous les niveaux de détails. Cette notion de niveau de détail est aussi appelée *niveau d'abstraction.*

Dans l'approche orientée par les objets, l'abstraction du monde réel est naturelle dans la mesure où les objets sont eux même vus comme une abstraction (représentation) des éléments du monde réel, le logiciel (abstraction "complète" du monde réel pour le problème considéré) étant un assemblage de tels objets.

Appliquer une politique d'abstraction basée sur les objets permet

– de faciliter la maintenance : en effet, la modification d'un logiciel est liée soit à une modification des caractéristiques des objets du monde réel manipulés dans le logiciel (donc la localisation de la modification est immédiate), soit une modification des fonctionnalités du logiciel (modification de l'abstraction); ces fonctionnalités étant rattachées á des objets, la portée de la modification est raisonnable;

3. Une abstraction représente les caractéristiques essentielles d'un objet qui le distinguent de toutes les autres sortes d'objets et ainsi en fournit les limites conceptuelles nettement définies, selon le point de vue de l'observateur.

– de faciliter la réutilisabilité : dans un domaine donné, une grande partie des objets modélisés sont sensiblement les mêmes d'un logiciel à un autre, même si certaines de leurs caractéristiques changent;

– de faciliter une approche protection de la confidentialité des informations, par le regroupement des traitements liés à ces informations ... et donc leurs évolutions.

7.4. Masquage / *Information Hiding*

Le principe du masquage peut être résumé ainsi : un élément logiciel ne doit voir que ce qu'il a besoin de voir ! Une information qu'il n'est pas utile de montrer ne doit pas être montrée. Il est possible de lui adjoindre une loi de l'em...nui maximum : si une information est visible, même si apparament elle n'est pas utile aux autres entités, il se trouvera toujours au moins une entité (laquelle ???) qui l'utilisera!

Au niveau des entités, le masquage est de deux formes :

– masquage de la réalisation d'une entité : il correspond à l'*information hiding*, et est très fortement lié à la notion de niveau d'abstraction (ou de détail);

– masquage de la présence d'entité(s) : il correspond à un graphe de visibilité réduit (donc correct) entre entités. Son obtention est liée à la mise en place d'une hiérarchie (cf hiérarchie).

Le masquage de la réalisation d'une entité suppose sa description en deux parties :

– l'*interface* d'une entité définit ce qu'il est nécessaire de connaître pour utiliser l'entité;

– son *corps* (ou le terme synonyme *réalisation*) définit sa structure interne, dont la connaissance est inutile pour une bonne utilisation de l'entité.

Information hiding :
The technique of encapsulating software design decisions in modules in such a way that the module's interfaces reveal as little as possible about the module's inner workings; thus, each module is a "black box" to the other modules in the system. The discipline of information hiding forbids the use of information about a module that is not in the module's interface specification. [ANSI/IEEE 729–1983]

G. BOOCH, utilise pour parler du même concept le terme *encapsulation*[4] :
Encapsulation is the process of hiding all of the details of an object that do not contribute to its essential characteristics [BOOCH–91]

L'encapsulation (au sens G. BOOCH ou information hiding ANSI/IEEE) est utile car :

– elle permet la plupart du temps une organisation plus claire (tous les niveaux de détail ne sont pas mélangés et utilisés), donc la mise au point;

4. Attention, ce terme a un sens particulier dans [ANSI/IEEE Std 729–1983] : Encapsulation :
The technique of isolating a system function within a module and providing a precise specification for the module.

– elle facilite l'indépendance des entités, donc leur évolution. En particulier, elle réduit certaines formes très dangereuses du couplage basées sur la connaissance interne de l'entité utilisée : ce type de couplage provoque de grosses difficultés pour la modification des entités donc la connaissance interne est utilisée dans d'autres entités : en effet, elle provoque des risques sérieux – et pas toujours connus car non commentés – de modifications sur ces entités.

De manière générale, le masquage améliore le temps de développement lié à la compilation (qu'elle soit séparée ou non) : si une information dont une entité ne se sert pas est vue par cette entité, toute modification de cette information provoquera une re–compilation de l'entité, donc a priori un nouvel exécutable, donc en théorie des tests de non–régression (unitaires, d'intégration ou de qualification suivant ce qui a déjà été réalisé) ... ce qui peut coûter très cher !

7.5. Hiérarchie */Hierarchy*

Une hiérarchie correspond à un graphe de dépendances sans cycle, c'est–à–dire qu'un supérieur dans une relation de dépendance ne peut se retrouver l'inférieur de son subordonné (donc la relation est non réciproque, ou plus mathématiquement : non symétrique).

A structure whose components are ranked into levels of subordination according to a specific set of rules. [ANSI/IEEE 729–1983]

L'inter–dépendance entre les éléments s'exprime par un ensemble de relations entre ces éléments. Ces relations peuvent être de différentes natures :

– relation "utilise"

– relation "est partie de" (décomposition)

– relation "est un" (instance); elle s'exprime entre un élément et un modèle d'éléments;

– relation "est un sous–ensemble de" (héritage)

Les cycles sont toujours un problème pour la mise au point d'un système (qu'il soit informatique, humain, electronique, mécanique ...) : ils sont donc à éviter, quelle que soit le type de relation. Pour certaines relations, le cycle n'a pas de sens, mais pour d'autres il est malheureusement utilisable et posera toujours un problème :

– si un cycle de la relation "est partie de" n'a pas de sens entre entités ("une entité A est partie d'une entité B, et B est partie de A"), il peut exister entre structures de données, et dans ce cas oblige à une gestion dynamique de la représentation;

– un cycle sur la relation "est un" ou "est un sous–ensemble de" n'a pas de sens (sauf pour exprimer ... l'équivalence, ce qui n'a aucun intéret en informatique);

Note : G. BOOCH limite le terme de hiérarchie au quatrième type de relation : Hierarchy is a ranking or ordering of abstractions. [BOOCH–91]

– le cycle le plus fréquent, car il a un sens, est celui lié à la relation "utilise" : "l'entité A utilise l'entité B, et B utilise A" : par exemple, A appelle des opérations de B et B appelle des opérations de A;
il est dangereux, car il pose des problèmes sérieux :
  * de mise au point : une erreur peut se révéler après plusieurs "tours" dans le cycle, d'où la difficulté de le faire apparaître; de plus, si une erreur existe dans un cycle, sa localisation est très délicate : qui est responsable? l'intégration partielle de plusieurs éléments d'un cycle n'est pas utile à cause des effets "action–réaction" liés au cycle : l'intégration doit donc se faire "d'un coup" entre tous les éléments du cycle;
  * d'évolution : toucher à un élément situé dans un cycle présente un risque important d'être amené à modifier les éléments du cycle et peut–être de reboucler plusieurs fois les modifications avant d'obtenir un nouveau cycle "stable".

Pour se convaincre du danger lié aux cycles, il suffit de faire un parallèle avec une organisation d'entreprise dans laquelle un cycle de responsabilité existerait ...

Les cycles sont donc à proscrire, et le fait d'établir une hiérarchie est une pratique saine et financièrement efficace.

Si une hiérarchie est sans cycle, ce n'est pas la seule caractérisation d'une hiérarchisation. Hiérarchiser un ensemble d'éléments, c'est aussi attribuer la responsabilité de la maîtrise des éléments.

Hiérarchiser une organisation d'éléments telle que la figure 7.2. a, c'est construire l'organisation de la figure 7.2. b ou 7.2. c :

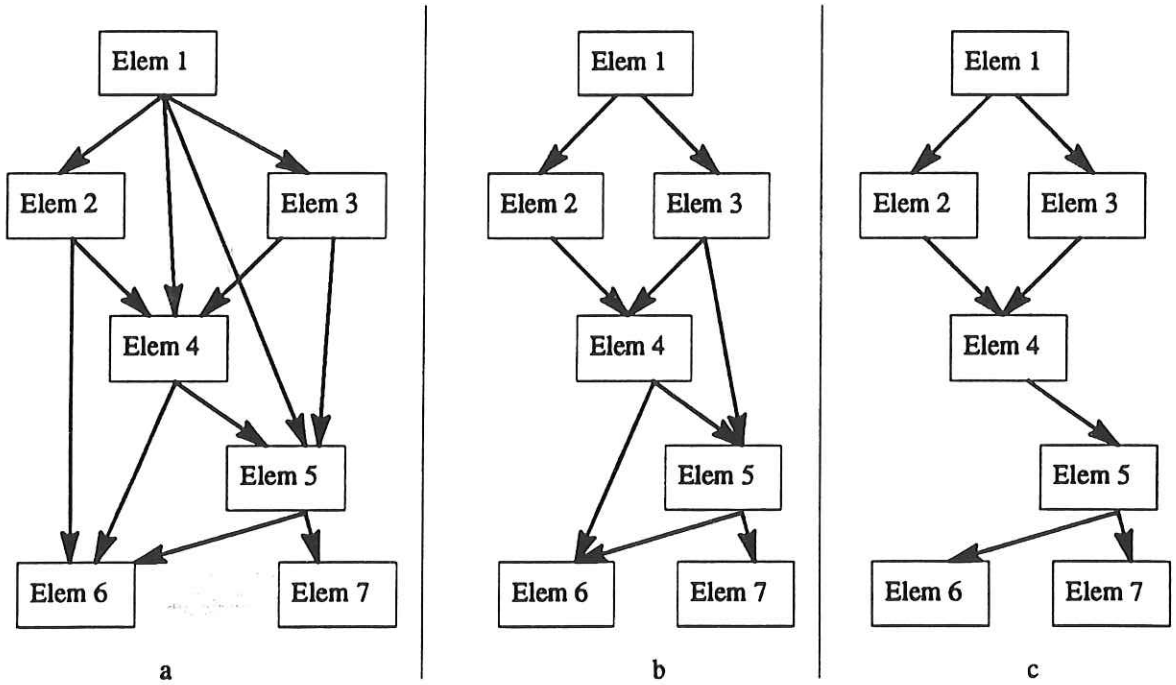


Figure 7.2.

C'est un mécanisme qui améliore le masquage au sens masquage de la présence d'entités.

Le masquage de la présence d'entité(s) correspond à rendre visible une entité A à une entité B si et seulement si B a (absolument) besoin de "voir" A : c'est l'optimisation (correspondant à une réduction) du graphe de visibilité entre entités.

Une entité qui voit beaucoup d'autres entités (grand fan–out) est souvent symptomatique d'une absence d'un tel masquage : cela correspond souvent à un mélange de niveaux de détails; une réorganisation basée sur la délégation de responsabilités entre entités permet souvent une nette amélioration de ce genre de masquage.

Un tel masquage associé à la hiérarchie améliore en particulier la mise au point en réduisant la complexité interne des entités (elles ne voient, donc ne gèrent pas trop de choses), ainsi que le coût des évolutions. D'autre part, les avantages généraux du masquage s'applique à cette forme de masquage obtenu par hiérarchisation.

7.6. Typage

A type is a precise characterization of structural and behavioral properties which a collection of entities all share. [ZILLES–84]

Pour la conception, l'aspect structurel est secondaire, et seul l'aspect comportemental est intéressant : un type est donc un modèle d'éléments ayant toutes les mêmes propriétés comportementales (ils sont capables de réaliser les mêmes opérations, ou, ce qui est équivalent, on peut leur appliquer les mêmes opérations).

Data type :
A class of data characterized by the members of the class and the operations that can be applied to them; for example, integer, real, logical. [ANSI/IEEE 729–1983]

Le mécanisme de typage permet de décrire non pas un élément, mais un ensemble d'éléments possédant le même ensemble de propriétés. C'est une première factorisation de description.

Associé à la notion de type, le mécanisme d'instanciation permet, à partir de la définition d'un type, de créer un ou plusieurs éléments définies selon le modèle qu'est le type : cela veut dire que ces éléments ont tous les propriétés qui ont été définies et associées au type. Ces éléments sont des *instances* (ou *occurences*) de ce type.

Par exemple, dans un langage de programmation avec types, déclarer une variable d'un type T correspond à réaliser une instance (instancier) de ce type.

L'intérêt de ces deux mécanismes est de permettre de définir *un* modèle possédant un ensemble de propriétés, puis des instances de ce modèle, plutôt que d'avoir à redéfinir toutes les propriétés pour chacune des instances du type : c'est une factorisation des définitions, qui induit

– une meilleure lisibilité : pour rechercher des objets liés par un même ensemble de propriétés, le réalisateur n'a pas à étudier chaque objet pour se rendre compte s'il possède tel ou tel ensemble de propriétés, mais seulement à regarder s'ils sont instances du même type; d'autre part, l'énoncé de ces propriétés est établi une seule fois, évitant d'"encrasser" le texte du programme d'une multitude de redites ;

– une meilleure évolutivité : en effet, faire évoluer un ensemble d'objets rattachés à une même définition de type est évidemment plus facile (et plus sûr) que d'avoir à rechercher tous les objets touchés par la modification et effectuer sur chacun d'eux cette modification; la cohérence de l'application vis à vis d'une telle modification est immédiate dans le cas d'une définition par type, alors que c'est un travail fastidieux et non sûr dans le cas de définition objet par objet;

– la possibilité de réutilisation, par essence même du principe de type.

Comme G. BOOCH, on utilisera de manière quasi synonyme les termes *classe* et *type*[5].

7.7. Parallélisme

Un aspect classique des systèmes temps réel est qu'il n'est pas toujours possible d'exprimer (facilement) un logiciel comme une séquence unique de traitements : le concepteur doit concevoir un logiciel dans lequel plusieurs activités peuvent "se passer en même temps". Il est alors nécessaire de déterminer le nombre d'activités simultanées, ainsi que leur description : cela correspond à la notion de processus parallèles (ou plus simplement processus).

Le parallélisme est la caractéristique d'un logiciel de décrire plusieurs séquences d'activités qui peuvent (réellement ou virtuellement) s'éxécuter simultanément.

Le terme *thread of control* correspond à l'identification d'une séquence d'activités.

Un processus (point de départ d'un *thread of control*) est l'identification d'une séquence d'activités qui s'exécutent simultanément (ou *en parallèle*) avec le reste des activités de l'application. Le parallélisme d'une application est donc identifié par les processus qui décrivent chacun une séquence d'activités.

Certains processus peuvent être rémanents (leur durée de vie est celle de l'exécution de l'application – on dit qu'ils sont *statiques*) d'autres sont temporaires (ils sont créés et disparaissent durant une partie de la durée de vie de l'application – on dit qu'ils sont *dynamiques*, car c'est l'exécution de l'application qui détermine l'instant de leur création).

Le parallélisme est *réel* si plusieurs processeurs fonctionnent simultanément, chacun étant associé à un processus et exécutant les activités liées à ce processus. Le parallélisme est *virtuel* si un seul processeur prend en charge les activités liées aux processus, mais dont le temps est partagé entre les processus : le concepteur conçoit le logiciel "comme si" plusieurs processus fonctionnaient simultanément.

5. Une classe modélise essentiellement un comportement, alors qu'un type est la plupart du temps lié à la notion de contrôle de type, c'est–à–dire à la non–interchangeabilité des éléments de deux types différents.
Une autre manière de présenter la différence est que deux ensembles d'éléments peuvent avoir la même abstraction (même ensemble de propriétés) pour le problème particulier auquel doit répondre le logiciel. Cependant, il peut s'agir de deux ensembles différents que le concepteur ne veut pas mélanger (les carottes et les navets) : dans ce cas, il introduira deux types, alors qu'en terme purement comportemental (dans le problème particulier) ils pourraient être représentés par une même classe.
Le terme classe est également la plupart du temps lié à la Programmation Par Objets (PPO) et au mécanisme d'héritage.

L'identification du parallélisme dans une application est un point délicat, car les performances de l'application en sont fortement dépendantes. Identifier plusieurs processus suppose en général de définir la *synchronisation* entre ces processus, de régler les problèmes de conflits d'accès simultanés (ou *concurrentiels*) sur des données ...

Le partage du temps entre les processus sur un processeur est réglé par un ordonnanceur, partie d'un exécutif temps réel qui propose un ensemble de mécanismes de synchronisation entre les processus.

Le parallélisme introduit un certain nombre de difficultés majeures :

- concevoir une application dans laquelle "plusieurs choses se passent en même temps et coopèrent" est difficile, donc source d'erreurs difficiles à appréhender;
- détecter une erreur dans une application parallèle est également difficile, car le jeu d'essai qui détecte l'erreur liée au parallélisme (sur la synchronisation ou sur les problèmes liés au partage de données) est à régler "au tournevis de très haute précision";

  de plus, le test de la synchronisation ne peut la plupart du temps que se faire sur la machine cible (sur laquelle on ne dispose pas toujours d'un équipement de test "confortable"), et est donc lié à l'intégration, donc est réalisé tard dans le cycle de développement d'un logiciel : les erreurs détectées alors sont un risque majeur pour les retards dans la livraison du logiciel;

  enfin, les tests du parallélisme ne sont pas toujours reproductibles, car l'erreur levée est parfois dépendante d'une situation temporelle extrêmement précise (l'arrivée d'une interruption 1ms – voire moins – avant ou après peut changer le comportement du logiciel) qui n'est pas reproductible avec la précision souhaitée;
- si une erreur est détectée, trouver d'où vient l'erreur est difficile (surtout si elle est non reproductible); la correction provoque la modification d'entités qui doivent être re–testés unitairement (non–régression), et l'intégration (et donc les tests qui y sont liés) doit être recommencée : le coût des tests est donc important; enfin, lorsqu'on passe le jeu d'essai qui a provoqué l'erreur, et si tout se pase bien, on ne sait pas si l'erreur est corrigée ou non, car la modification a pu modifier de manière "epsilonesque" le partage du temps et donc ne plus réveler un défaut qui peut toujours exister.

Le parallélisme représente donc un risque technique majeur dans le développement des logiciels, et il faut donc être particulièrement vigilant sur son expression : il est identifié en conception préliminaire, les mécanismes de synchronisation et les partages d'informations sont explicitement indiqués dès cette phase. Savoir si une entité "contient" une activité parallèle ou non est une information importante. Les entités qui représentent des éléments dont le partage est critique (c'est–à–dire qui contiennent des données partagées par plusieurs processus) seront séparés des entités non critiques (ceux dont le partage par plusieurs processus ne représente pas un risque : les modèles d'entités par exemple, car ils ne contiennent pas de données).

8. GLOSSAIRE

8.1. ***Abstract type*** (cf 8.46.)

8.2. **Abstraction / *Abstraction*** (cf 7.3.)

L'abstraction est l'opération intellectuelle qui permet de définir un objet par l'ensemble des propriétés qui le caractérisent dans un domaine donné. La notion de domaine correspond à la notion de point de vue.

(1) A view of a problem that extracts the essential information relevant to a particular purpose and ignores the remainder of the information.
(2) The process of forming an abstraction. [ANSI/IEEE 729–1983]

An abstraction denotes the essential characteristics of an objet that distinguish it from all other kinds of objets and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.[1] [BOOCH–91]

8.3. **Actif / *Active*** (cf 8.37.)

8.4. **de Base / *Basic*** (cf 8.10.)

8.5. **Classe d'objets / *Object classe*** (cf 3.2.2.)

> Une *classe d'objets* (ou plus simplement : une classe) est un ensemble d'objets qui ont une structure et un comportement communs. [BOOCH–91]

Un objet qui est conforme au modèle qu'est la classe est une *instance* de la classe.

8.6. **Cohésion / *cohesion*** (cf 8.29.)

8.7. **Collection / *Collection*** (cf 8.29.)

8.8. **Collection / *Collection*** (cf 3.5.2. & 4.7.)

La *collection* est un regroupement de modèles (types abstraits ou génériques).

8.9. **Comportement d'un objet / *Object behaviour*** (cf 3.2.1.)

> Le *comportement* définit comment un objet agit ou réagit, en termes de changements d'états et d'échanges de messages. [BOOCH–91]

Une *opération* est une action qu'un objet "lance" sur un autre objet en vue de provoquer une réaction. Echange de message et appel d'opération sont synonymes.

1. Une abstraction représente les caractéristiques essentielles d'un objet qui le distinguent de toutes les autres sortes d'objets et ainsi en fournit les limites conceptuelles nettement définies, selon le point de vue de l'observateur.

8.10. Composite–de–base / *Composite–basic* (cf 3.3.3.)

Un objet est *composite* s'il se décompose en d'autres objets et si sa description se décompose en la description des objets contenus (donc l'entité qui le décrit se décompose en entités qui décrivent chacun l'un des composants de l'objet).

Un objet *de base* est un objet non composite, c'est–à–dire que l'entité qui le décrit ne se décompose pas en (sous–)entités.

8.11. Conception / *Design*

Activité créatrice qui, partant des besoins exprimés, des moyens existants et des possibilités technologiques, aboutit à la définition d'un produit satisfaisant ces besoins et industriellement réalisable. [NF L 00–007]

Démarche de définition des tâches à accomplir et de résolution des problèmes dans le but de créer un produit ou un service destiné à satisfaire des besoins exprimés. Le terme "conception" désigne également le résultat des opérations précitées. [AQAP 15]

(1) The process of defining the software architecture, components, modules, interfaces, test approach, and data for a software system to satisfy specified requirements[1].
(2) The result of the design process[2]. [ANSI/IEEE, Std 729–1983]

8.12. Conception détaillée / *Detailed design*

Ensemble des activités consistant à détailler les résultats de la conception préliminaire, tant sur le plan algorithmique que sur celui de la structure des données, jusqu'à un niveau suffisant pour permettre le codage [Z61–102]

(1) The process of refining and expanding the preliminary design to contain more detailed descriptions of the processing logic, data structures, and data definitions, to the extent that the design is sufficiently complete to be implemented.
(2) The result of the detailed design process. [ANSI/IEEE, Std 729–1983]

Le terme n'est pas défini directement dans la DOD–STD–2167A, mais l'ensemble des activités qu'il désigne recouvre également des activités de gestion de projet, de gestion des configurations, d'ingénierie (en particulier identification des CSUs[3], affectation des exigences, description des interfaces, organisation et description des tests d'intégration des CSCs et des tests unitaires de CSUs), de test de qualification et d'évaluation. cf [DOD–STD–2167A] ch. 5.4, pour plus de précisions.

1. Le processus de définition de l'architecture, des composants et des modules logiciels, de leurs interfaces, de l'approche de test et des données d'un système logiciel qui satisfont les exigences spécifiées
2. Le résultat du processus de conception.
3. Unités logiciel

8.13. Conception préliminaire / *Preliminary design*

Ensemble des activités conduisant à l'élaboration de l'architecture du logiciel. [Z61–102]

(1) The process of analyzing design alternatives and defining the software architecture. Preliminary design typically includes definition and structuring of computer program components and data, definition of the interfaces, and preparation of timing and sizing estimates.
(2) The result of the preliminary design process. [ANSI/IEEE, Std 729–1983]

Le terme n'est pas défini directement dans la DOD–STD–2167A. La définition la plus proche est celle ANSI/IEEE. L'ensemble des activités que le terme désigne, recouvre des activités de gestion de projet, de gestion des configurations, d'ingénierie (en particulier identification des CSCs[4], affectation des exigences, description des interfaces, définition des exigences de test des CSCs), de test de qualification et d'évaluation. cf [DOD–STD–2167A] ch. 5.3, pour plus de précisions.

8.14. Couplage / *Coupling* (cf 8.29.)

8.15. Décomposition / *Decomposition* (cf 7.1.)

Processus qui permet de décrire une entité comme étant constituée de sous–entités, et définissant les inter–dépendances entre ces entités.

8.16. *Design* (cf 8.11.)

8.17. *Detailed design* (cf 8.12.)

8.18. Elément / *Element* (cf 3.1.)

Terme générique correspondant à tout élément ou partie d'élément (du monde réel, du monde informatique), clairement identifiable, c'est–à–dire :

- possédant un nom propre,
- dont on peut caractériser les limites/frontières, c'est–à–dire ce qui en fait partie par rapport à ce qui n'en fait pas partie,
- qui est manipulable de manière indépendante.

8.19. *Encapsulation* (cf 8.29.)

8.20. Entité / *Entity* (cf 3.5.1.)

Dans l'approche COO, une *entité* est un élément qui est le support de description d'un objet si celui–ci a une description propre, ou à la description d'un modèle d'objets (classe ou générique) ou d'une collection. Elle correspond donc à une unité de programme Ada, ou à un ensemble de telles unités.

Une entité est décrite en deux parties, son *interface et son corps*. (cf masquage)

4. Composants logiciels

8.21. Entités (généralités sur) / *Entities (overview)* (cf 3.5.)

La figure suivante établit la correspondance entre les entités et les objets/modèles qu'elles implémentent.

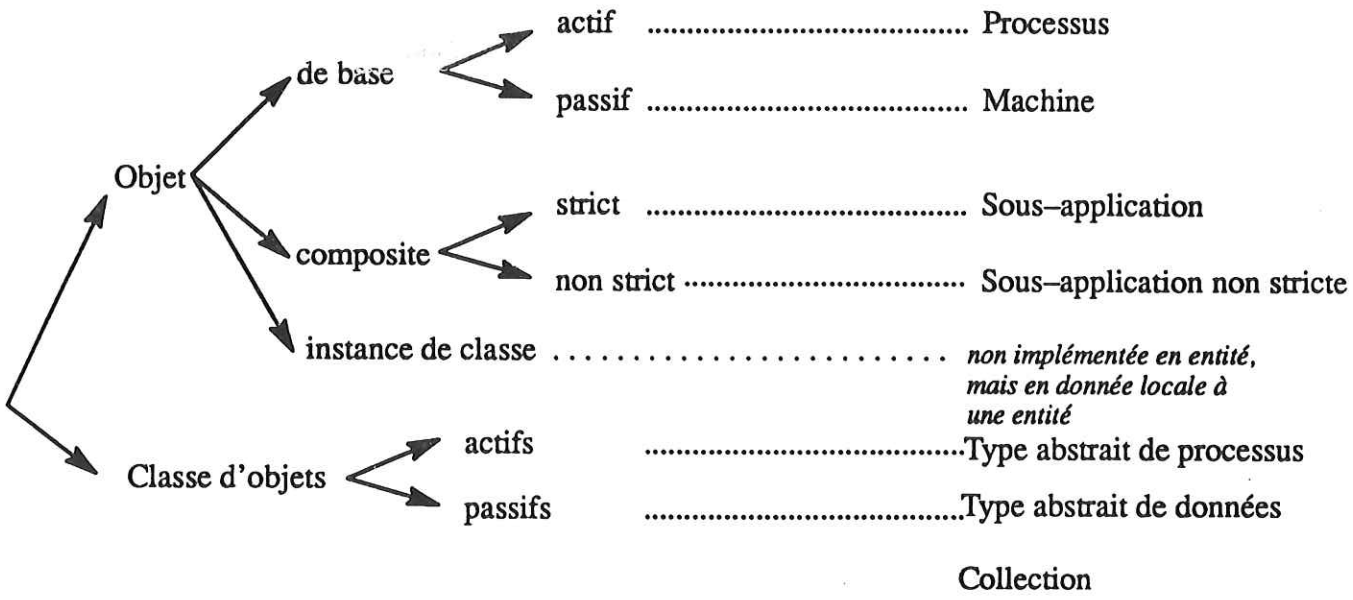
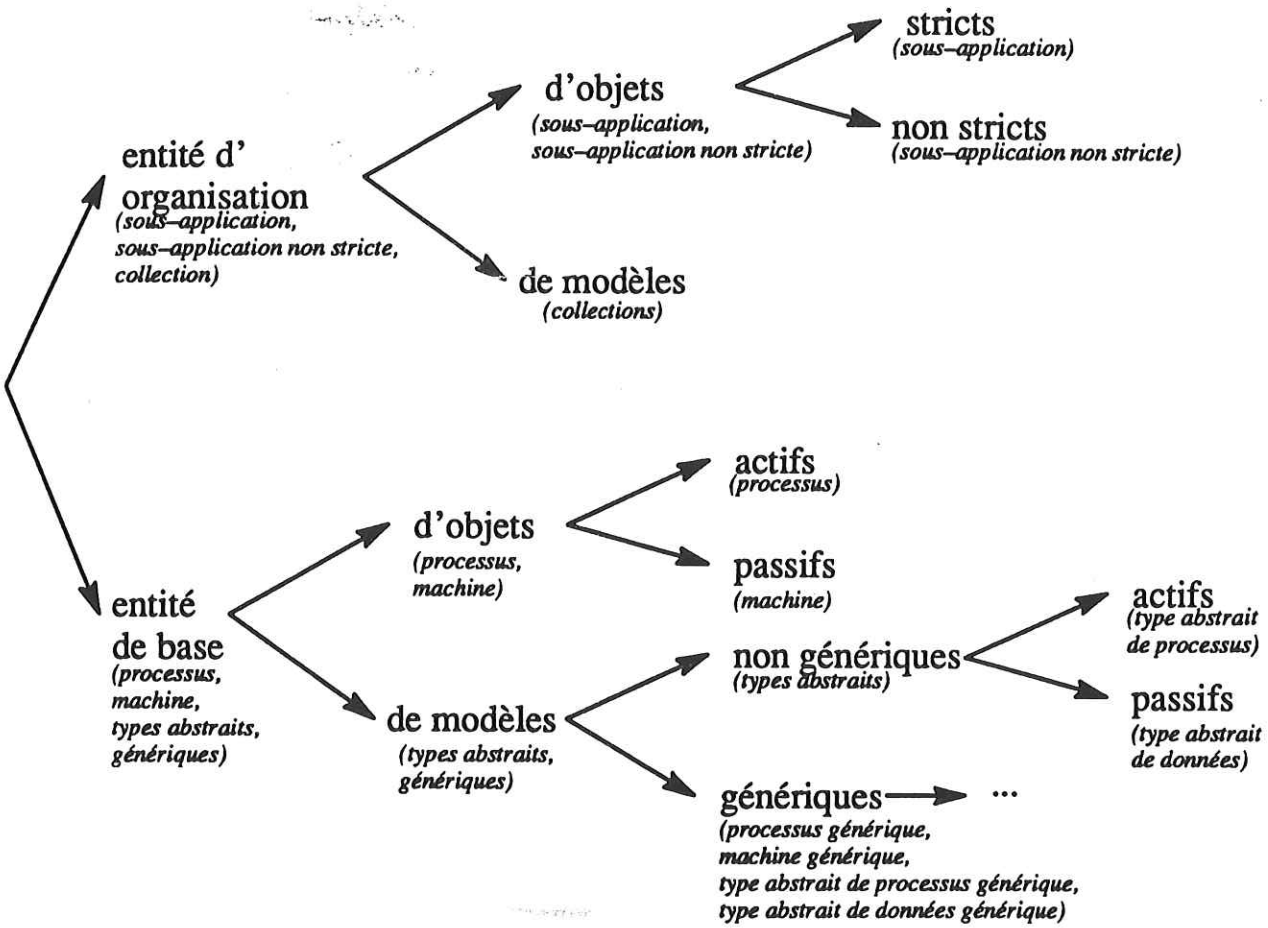


Figure 8.1.

La vision duale de l'ensemble des entités est la suivante :


entité d'
organisation
(sous–application,
sous–application non stricte,
collection)
d'objets
(sous–application,
sous–application non stricte)
stricts
(sous–application)
non stricts
(sous–application non stricte)
de modèles
(collections)
entité
de base
(processus,
machine,
types abstraits,
génériques)
d'objets
(processus,
machine)
actifs
(processus)
passifs
(machine)
de modèles
(types abstraits,
génériques)
non génériques
(types abstraits)
actifs
(type abstrait
de processus)
passifs
(type abstrait
de données)
génériques
(processus générique,
machine générique,
type abstrait de processus générique,
type abstrait de données générique)

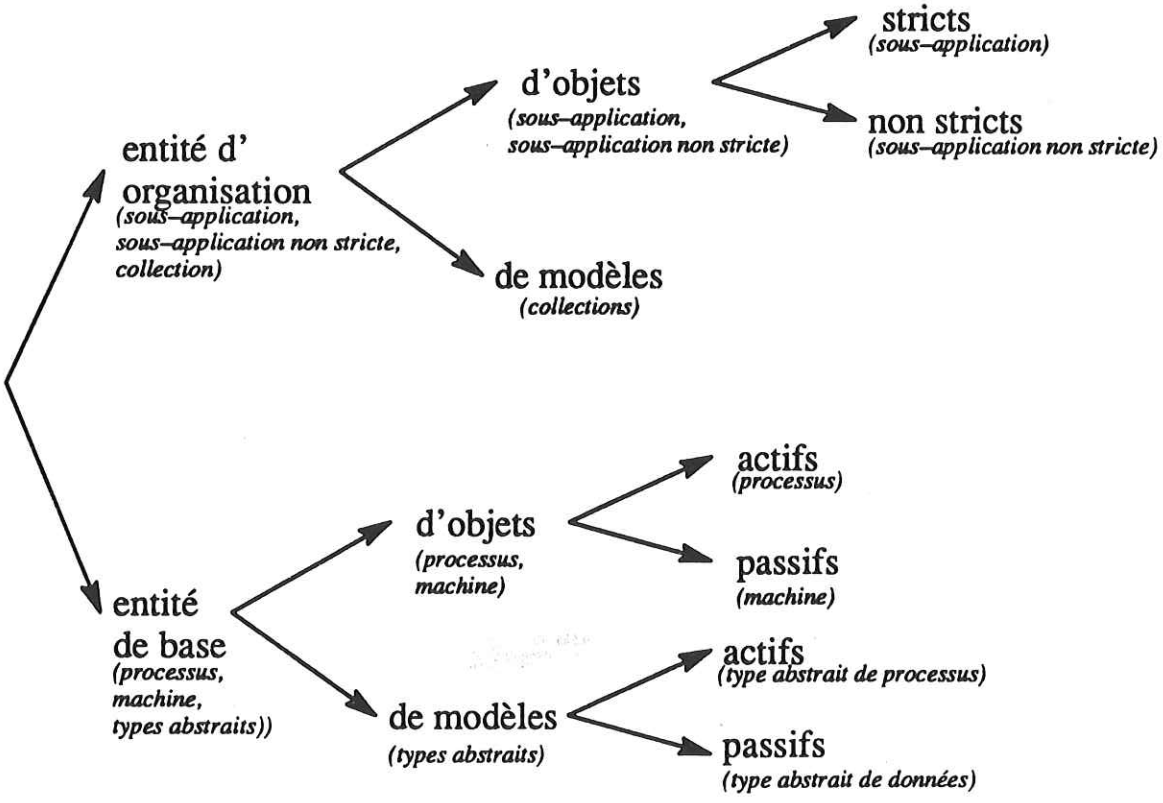

entité d'
organisation
(sous–application,
sous–application non stricte,
collection)
d'objets
(sous–application,
sous–application non stricte)
stricts
(sous–application)
non stricts
(sous–application non stricte)
de modèles
(collections)
entité
de base
(processus,
machine,
types abstraits))
d'objets
(processus,
machine)
actifs
(processus)
passifs
(machine)
de modèles
(types abstraits)
actifs
(type abstrait de processus)
passifs
(type abstrait de données)

Figure 8.2.

8.22. Etat d'un objet / *Object state* (cf 3.2.1.)

L'*état* d'un objet englobe toutes les propriétés (en général statiques) d'un objet, plus les valeurs courantes (en général dynamiques) de chacune de ces propriétés. [BOOCH–91]

Le terme propriété signifie caractéristique.

La *structure* d'un objet est la représentation concrète de son état. [BOOCH–91]

8.23. Génériques[5] / *Generic* (cf AUCUN LIEN)

Un *objet générique* est un modèle paramétrable d'objets, c'est–à–dire :

– qu'il est possible de faire des instances d'un objet générique;

– que cette instanciation est paramétrable.

L'*instanciation d'un objet générique* complète la définition et fournit donc soit un objet (si la définition est totalement complétée) soit un objet générique plus défini que le modèle initial (si la définition est partiellement complétée).

Une *classe générique* est un modèle paramétrable de classe, c'est–à–dire :

– qu'il est possible de faire des instances d'une classe générique : ces instances sont des classes;

– que cette instanciation est paramétrable.

L'*instanciation d'une classe générique* complète la définition et fournit donc soit un modèle complètement défini d'objets (si la définition est totalement complétée) soit une classe générique plus définie que le modèle initial (si la définition est partiellement complétée).

8.24. Hiérarchie / *Hierarchy* (cf 7.5.)

Une hiérarchie correspond à un graphe de dépendances sans cycle, c'est–à–dire qu'un supérieur dans une relation de dépendance ne peut se retrouver l'inférieur de son subordonné (donc la relation est non réciproque, ou plus mathématiquement : non symétrique).

A structure whose components are ranked into levels of subordination according to a specific set of rules. [ANSI/IEEE 729–1983]

8.25. Identité d'un objet / *Object identity* (cf 3.2.1.)

L'*identité* est la propriété d'un objet qui le distingue de tous les autres objets. [BOOCH–91]

5. les termes objets génériques et classe génériques sont en fait impropres : il vaudrait mieux parler de générateurs d'objets et générateurs de classes. Ils sont malheureusement conservés car ils font partie du langage courant, bien qu'un objet générique ne soit pas un objet et qu'une classe générique n'est pas réellement une classe ! En particulier un objet générique est plus proche d'une classe (classe paramétrable) que d'un objet !

8.26. ***Information hiding*** (cf 8.28.)

8.27. **Machine / *Machine*** (cf 3.5.2. & 4.2.)

La *machine* est l'implémentation d'un objet qui est passif, c'est–à–dire qui ne sait que répondre à des stimulis.

8.28. **Masquage / *Information hiding*** (cf 7.4.)

Information hiding :
The technique of encapsulating software design decisions in modules in such a way that the module's interfaces reveal as little as possible about the module's inner workings; thus, each module is a "black box" to the other modules in the system. The discipline of information hiding forbids the use of information about a module that is not in the module's interface specification. [ANSI/IEEE 729–1983]

G. BOOCH, utilise pour parler du même concept le terme *encapsulation*[6] :
Encapsulation is the process of hiding all of the details of an object that do not contribute to its essential characteristics [BOOCH–91]

8.29. **Modularité / *Modularity*** (cf 7.2.)

Les caractéristiques du logiciel qui présentent une structure en modules[7] très indépendants les uns des autres. [VQ 02–1990 E/F]

The extent to which software is composed of discrete components[8] such that a change to one component has minimal impact on other components. [ANSI/IEEE 729–1983]

Le *couplage* caractérise le niveau de dépendance entre entités.

La *cohésion* caractérise le niveau de dépendance entre les composants d'une entité.

8.30. **Objet / *Object*** (cf 3.2.1.)

Un objet a
- un *état*,
- un *comportement*,
- une *identité*. [BOOCH–91]

La structure et le comportement d'objets semblables sont définis dans leur classe

6. Attention, ce terme a un sens particulier dans [ANSI/IEEE Std 729–1983] : Encapsulation :
The technique of isolating a system function within a module and providing a precise specification for the module.

7. le terme entité de ce document est synonyme à cette notion de module

8. le terme entité est synonyme à cette notion de composant

commune[9].

Un objet est une abstraction du monde réel.

8.31. ***Object behaviour*** (cf 8.9.)

8.32. ***Object classe*** (cf 8.5.)

8.33. ***Object identity*** (cf 8.25.)

8.34. ***Object state*** (cf 8.22.)

8.35. **Parallélisme / *Parallelism*** (cf 7.7.)

Le parallélisme est la caractéristique d'un logiciel de décrire plusieurs séquences d'activités qui peuvent (réellement ou virtuellement) s'éxécuter simultanément.

Le terme *thread of control* correspond à une séquence d'activités.

Un *processus* (point de départ d'un *thread of control*) est l'identification d'une séquence d'activités qui s'exécutent simultanément (ou *en parallèle*) avec le reste des activités de l'application. Le parallélisme d'une application est donc identifié par les processus qui décrivent chacun une séquence d'activités.

8.36. **Partagé / *shared*** (cf 3.3.2.)

Un objet est partagé si ses opérations peuvent être appelées par plusieurs objets actifs[10].

8.37. **Passif–Actif / *Passive–Active*** (cf 3.3.1.)

Un objet est passif si son activité est uniquement composée de réactions provoquées par les appels d'opérations effectués par son environnement.

Un objet est actif s'il a une activité simultanée (ou parallèle) avec le reste de l'application, en dehors des réactions provoqués par les appels d'opérations effectués par son environnement. Un objet actif est dit avoir un *comportement propre*.

8.38. ***Preliminary design*** (cf 8.13.)

8.39. **Processus / *Process*** (cf 3.5.2. & 4.3.)

Le *processus* identifie un (et un seul) *thread of control*, c'est à dire une séquence d'activités qui s'exécutent simultanément (virtuellement ou réellement) avec les autres séquences d'activités.

9. On ne reprend pas la définition complète de G.BOOCH qui ajoute : les termes instance et objet sont synonymes.
En effet, il part du principe que tout objet appartient à une classe, ce qui permet d'unifier les définitions. Dans cette approche, si un objet est unique dans une application, il est assez fastidieux de décrire la classe, puis d'indiquer que cet objet est l'instance de la classe. En fait, ce qui est proposé est de décrire directement l'objet, mais "conceptuellement", cela correspond à décrire une classe anonyme dont l'objet est l'unique instance.
Ce genre d'argumentation est intéressante d'un point de vue théorique, mais n'apporte rien au savoir–concevoir, et n'a donc pas été introduite ici. De plus, cela peut amener des confusions sur le partage de telles informations, dans un contexte parallèle.

2. Plus précisément, un objet est partagé si ses opérations peuvent être appelées par plusieurs objets actifs de base, ou encore s'il peut être "traversé" par plusieurs *thread of control*.

8.40. Relations / *Relationships* (cf 3.4.)

Les relations sont des trois genres suivants :

– relation "utilisation"

– relation "est partie de" (décomposition)

– relation "est un" (ou "est instance de")

Les relations introduites ci–après s'appliquent aussi bien aux objets, classes d'objets et génériques qu'aux entités qui les décrivent.

8.41. *Shared* (cf 8.36.)

8.42. Strict – non strict / *strict – non strict* (cf 3.3.4.)

Un objet composite est *strict* si l'objet composite ne contient ni partie de structure ni comportement qui ne soit attaché à l'un des objets composants.

Un objet composite est *non strict* si une partie de la structure et/ou du comportement de l'objet composite n'est pas défini par l'un des objets composants mais est lié à l'objet "assemblage".

8.43. Structure / *Structure* (cf 8.22.)

8.44. Sous–application / *Sub–application* (cf 3.5.2. & 4.4.)

Une *sous–application* est un assemblaget d'entités (pas de contenu autre que les entités). Elle comprend deux formes :

- la forme stricte (une telle sous–application est simplement appelée "sous–application") : le corps de la sous–application (stricte) ne contient que la description de ses entités composantes et de leurs interfaces, ainsi que l'attachement de chaque opération de la sous–application à une opération de l'objet composant qui prend en charge l'opération;
- la forme non stricte (une telle sous–application est appelée "sous–application non stricte") : une *sous–application non stricte* est un assemblage non strict d'entités (une partie de son contenu ne se retrouve dans aucune des entités qui le composent). Le corps de l'objet non strict contient donc la description de ses entités composantes et de leurs interfaces, ainsi que les éléments de structure qui n'appartiennent à aucun composant, et les opérations qui correspondent en général à une logique d'appel des opérations des objets composants.

8.45. Typage / *Typing* (cf 7.6.)

A type is a precise characterization of structural and behavioral properties which a collection of entities all share. [ZILLES–84]